



The Undocumented Internals of the

 **bitcoin,**



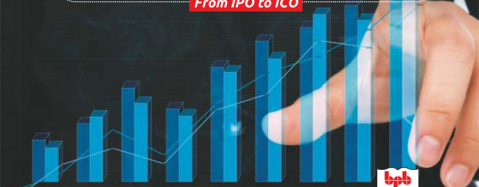
ethereum

AND



blockchains

From IPO to ICO



VIJAY MUKHI

FOREWORD BY
NITIN KHANAPURKAR

BPB PUBLICATIONS

*The
Undocumented Internals of the
bitcoin,
ethereum
and
blockchains*

by
Vijay Mukhi



BPB PUBLICATIONS

FIRST EDITION 2018

Copyright © BPB Publications, India

ISBN : 978-93-8655-130-6

All Rights Reserved. No part of this publication may be reproduced or distributed in any form or by any means or stored in a database or retrieval system, without the prior written permission of publisher with the exception that the program listings may be entered, stored and executed in a computer system, but they may not be reproduced by publication.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true and correct and the best of author's & publisher's knowledge. The author has made every effort to ensure accuracy of this publications, but cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj,
NEW DELHI-110002
Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE

376, Old Lajpat Rai Market,
DELHI-110 006,
Ph: 23861747

DECCAN AGENCIES

4-3-329, Bank Street,
HYDERABAD-500195
Ph: 24756967/24756400

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002

Dedication

*This book is dedicated to my friend, the late **Shri Anil Ganguly**.
His journey of life and his success stories have inspired us in a big way.*

*The legendary director had only one advice for all:
Never Give Up On Your Dreams, no matter what.*

Foreword

The Future of VALUE EXCHANGE

The “Fintech” sector has been a passionate advocate for Bitcoins. It is touted by early adopters to be the silver bullet that will take out intermediaries like banks and governments from the financial ecosystem and make exchange of value much more efficient. On the other hand, some financial veterans dismiss the entire concept as a fad.

The buzz around Bitcoins just keeps getting louder and louder.

Bitcoins provide a barrier against governments that control central banks who in turn control fiat money. The recent demonetization initiatives in countries like India and Venezuela have woken up citizens to the need for a currency that is not controlled by governments. Cryptocurrencies seem to be an ideal answer to this need.

On the dark side, Bitcoins have had a notorious history so far. They have been used for illegal purposes like purchase of drugs off the dark web. However, Bitcoins have managed to survive that phase and are now on the cusp of becoming a mainstream medium of value exchange. Large financial organizations like BNP Paribas, Citi Bank, Barclays etc. have Bitcoin initiatives. (<http://www.coindesk.com/8-banking-giants-bitcoin-blockchain/>)

The anonymity of the owner of Bitcoins is a boon for some and certainly a bane for those who want to track it down.

Bitcoin has reached an impressive milestone in the first week of December, 2017, where the popular cryptocurrency became more valuable than gold for the first time ever.

The blockchain-powered currency topped off gold, hitting the price of \$16000 for a single unit of Bitcoin compared to \$1,233 for a troy ounce of gold. (Ref: BBC)

Recovering from a sharp fall to \$200 per unit in mid-2015, Bitcoin has experienced a rather significant surge throughout the latter part of 2017 and early 2018 so far. Gold on the other hand has circled around the \$1,200 price point for a while now, with some sporadic hikes and drops here and there. This could be too good to be true for some people while the virtual world believes this is just a beginning.

The Bitcoin is currently heavily managed / controlled by the miners from China. This certainly poses the challenge in the way it will be adopted by the World. We will certainly see a huge amount of churn happening in times to come. The lack of regulations and fundamentals of valuation raises another concerns about longevity and stability. However it is certain that an ecosystem will evolve which will depolarize the Bitcoin and we will see regulations evolving around the rise of crypto currency

However, Bitcoin is just one of the applications of a much bigger technology viz. blockchain. Bitcoin is to Blockchain as jewelry is to gold. A blockchain is a distributed database that maintains a continuously growing list of ordered records called blocks. Each block contains a timestamp and a link to a previous block. By design, blockchains are inherently resistant to modification of the data — once recorded, the data in a block cannot be altered retroactively. Blockchains are “an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way. The ledger itself can also be programmed to trigger transactions automatically.” (Reference: [https://en.wikipedia.org/wiki/Blockchain_\(database\)](https://en.wikipedia.org/wiki/Blockchain_(database)))

The Blockchain is actually the building block of any application involving value exchange without involving an intermediary. This not only about the Fin Tech or changing the Financial Ecosystem but this is about changing the way Value is stored and exchanged.

The Blockchain has the potential to disrupt any industry that involves exchange of value.

Some of the sectors that could go mainstream with blockchain technology are:

- Banking
- Insurance
- Public sector
- Real estate
- Supply Chain / Logistics

The blockchain technology seems to have a bright future, there are significant challenges to be overcome before we can expect to see greater adoption.

- **Control, Security and Privacy:** These concerns need to be addressed before users put their faith in blockchain solutions. Security incidents where entire exchanges have been hacked and Bitcoins stolen, do little to give users confidence. Further, blockchain transactions are pseudonymous rather than being anonymous. Hence it is possible to trace the real identity of a user.
- **National regulations:** Each country has a differing approach towards Bitcoins. Some are hostile towards it while many are silent. Due to this, existing financial institutions may find it difficult to adopt blockchains and Bitcoins.
- **Mining power concentrated in the hands of few:** The huge power and accompanying capital investment has resulted in Bitcoin mining power being concentrated in the hands of few. This has led to a heavily skewed system that was initially built to be open and peer based.
- **Scalability:** Blockchains do not offer the same transaction processing speeds as those of other payments systems. Hence, when there is wide-scale adoption of blockchains, real time settlement of transactions could be a problem.
- **High initial capital costs:** While blockchains lower per transaction costs, implementing a blockchain solution is an expensive proposition.
- **Awareness:** Blockchains and Bitcoins are still in the “Nerd” domain. Unless the general public become aware about and start adopting them, it will be difficult to achieve a critical mass.

Despite the misgivings, the Blockchain technologies have been gaining greater traction. Organizations and individuals cannot remain mere bystanders given the potential blockchains have to cause disruption in almost every industry. Hence, programmers would do well to understand the finer aspects of this technology.

Vijay Mukhi's book takes a close look at the Blockchains and Bitcoins to understand their inner workings. It demystifies the nuts and bolts of the Blockchain and explains every bit of the block. Using programming languages like Python and Go, it sets out on an adventure to decipher each element in the blockchain with a view to use these elements when building blockchain applications. It tackles difficult concepts like encryption, Segregated Witnesses, Merkle Hashes etc. using a unique mix of programmes and witty commentary. Of course, his unique style and the actual code decoding the Blockchain is huge contribution to the Virtual Technology world.

Vijay Mukhi has done a great job in lifting the veil off blockchains and Bitcoins to give us an intimate peek into this exciting emerging technology.

Nitin Khanapurkar

Global Head of Governance Apex Fund Services based in London and before he served as partner with KPMG and Deloitte.

Nitin Khanapurkar

About the Author

Vijay Mukhi will be 61 when the year 2018 ends which makes him an elder statesman. He has basically done two things in his life. Teaching at his own computer training institute called Vijay Mukhi's Computer Institute (VMCI). At the same time, he wrote lots of books, over 80 of them ranging from traditional programming languages like C, C++, Autodesk Animator, Lotus 123, Java, C# Odyssey etc.

His technical books have been translated into languages like Japanese and Portuguese. All the books are written in the way he teaches. For him, every line of code is worth a thousand words, so he explains every program, line by line. Vijay Mukhi is not a software developer but a teacher at heart.

Earlier Vijay would write a book within 100 days. Now he takes two to three years to finish a book. And, when he is not writing or teaching, he is seen working with the law enforcement agencies and the industry helping them track down the cyber-criminal. He gives lectures on the latest developments in technology. His passion is cracking difficult and complex code and then simplifying it.

His email address is vijaymukhi712@gmail.com and his Bitcoin address is

1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv.

Acknowledgements

In the three years of making of this book, many events have occurred. The versions of source code, functions, libraries modules and many more things have changed. All in all, the book is ready to go. I want to thank a few people who have always been there by my side.

My publisher friend, Manish Jain of BPB Publications: He has trusted me blindly and published all my books and promises to do so further.

My friends Harish Mehta and Raj Saraf, who have always encouraged me and shown great interests in all my demos of newer technologies.

Nitin Kanapurkar, a friend again who spent time to write the forward. His business acumen is fantastic.

Sonal, my wife who has always been there with me in my journey of learning and life. She has quietly taken all my optimisms in stride.

My family, especially Dimple. Her positivity and cheering nature has always brought in freshness to life.

And finally, a big thank you to all the coders, developers, programmers, netizens, who uploaded all their work and helped me understand this subject better. Without the source code and Google, this journey would have been a struggle and difficult one. So once again, I acknowledge all the intelligent and smart ones out there for starting this new trend in technology and business.

Introduction

Me: Hi, my name is Vijay Mukhi.

You: Wake up Vijay and smell the coffee. You are not in your classroom, you are writing an introduction to your book.

Me: You cannot teach an old dog any new tricks.

You: Which only means that this book is identical to the 80+ books you have published in the last 30+ years.

Me: Yes, why change a winning formula.

You: Why should anyone buy your book, there are a million books on Bitcoins, Ethereum and the Blockchain.

Me: That's 100% true.

You: So, what makes your book different from the others in the market?

Me: Two reasons. First, my book does not use words to explain Bitcoins and more. There is very less theory. It uses code. It is a technical book for programmers. In my class, when I teach, I don't use words to explain a concept. I write one line of code and explain it. Then I write another and explain both. I have learnt, over the years, that nobody understands a new concept in the first round ever. Therefore, I repeat myself in my class and I have used the same approach in this book. Secondly, the code in the book is explained one line at a time. The same code is also explained differently throughout the book at different stages. All in all, we have explained 322 programs mostly written in Python that contain 2898 lines and 19977 characters.

You: Since most of the code is in Python, it could only mean that the main hero in your book is Python.

Me: Well there is C++ and Golang and very little JavaScript/Solidity also.

You: A million-dollar question, why should I buy your book.

Me: I started writing this book in the year 2015, finished in the year 2017. Amongst all the books I have written so far, this is the most difficult book. Bitcoin is at the confluence of Programming, Mathematics and Cryptography.

You: We all know how difficult it is to define a Bitcoin and a Blockchain, but ...

Me: A dream of mine was to write a book with the title containing the words undocumented and internals. It's been a dream come true.

You: Seriously?

Me: When you download the Bitcoin Core software on your computer, lots of files and folders get created on your machine. I have attempted to document each file and folder that gets created. As an example, the UTXO (whatever it means) is one of the most important artifacts created by the Bitcoin Core software. The only documentation of this Nobel prize winning Bitcoin invention can be found in some comments in the C++ Bitcoin source code.

You: By reading the source, you have explained the format and usage of every Bitcoin file and folder. Sound's good. But, why the step motherly treatment to Ethereum, only three chapters.

Me: Ethereum demands a book of its own. I just wanted to contrast a few differences between two of the most important blockchain technologies, Bitcoin and Ethereum.

You: How much of this book is original material.

Me: Very difficult question. When I started writing books 30 years ago, every book published was original. The reason being, there was no internet so, there was very limited or no access to other people's work. I now admit, that my thinking process has been largely influenced by what I read on the internet. So, by and large, some of the code in my book is derived from code snippets that I found on various forums. They are spread out throughout the book to explain certain concepts without which the book would not have been complete. But, the largest contributor to this book is the source code and the comments within it. The original source code is written by some of the finest minds in the world. Therefore, my book is dedicated to the 100+ programmers who have written the Bitcoin source. These are the geniuses who have kept the Bitcoin ecosystem flying high. I would like people to freely use code written in this book for whatever purpose they want. I also read the pull requests which are created when programmers make changes to the code.

You: What beef do you have against grammatical checkers.

Me: Lots. For example, Microsoft Word changes the word `ascii` to `ascii`. The code written in Python was treated at times as German, Italian, Dutch etc. At present, there are no grammatical checkers that understand code.

You: I feel that you are angry about the direction Bitcoin has taken.

Me: Yes. Ethereum, Zcash, Monero, Ripple etc., came much later to the crypto currency party. But, they are seen to be using better technology. The single largest infusion of technology into Bitcoins is called Segregated Witness. We may not experience Segregated Witness for another decade or so or maybe never. It's not because of bad technology but the way changes can be introduced into Bitcoins. Over 95% of the miners must agree or it's a no-go. Today, its about 30% who agree. Which means that the developers have no control over what code makes up Bitcoins, the businessmen or men in suites or the guys with raw processing power decide. In our view, the decision must be taken by the people who write code. There is no right solution to this block scaling problem, but anything will be better than what we are using today.

You: How many Bitcoin forks do you see in the future.

Me: We have seen two forks as of October 2017, these are the original Bitcoin Core and Bitcoin Cash. In November, I believe that segwit2x will hard fork and we will have 3 Bitcoins, Ethereum has two. The number of hard forks will jump to a 100, Bitcoin finally has its mojo back.

You: What about Bitcoins market cap.

Me: Its touching 100 billion dollars and the price of Bitcoin in December 2017 crossed 16000 dollars. When we started writing this book it was a measly 500 dollars. We are not kicking ourselves for buying any Bitcoins then and have no plans for buying any Bitcoins today. This book will indirectly allow you to predict the price of Bitcoins by understanding the magic sauce that drives Bitcoins.

You: Therefore, this book is not up-to-date.

Me: No, and that is why we did not release this in January 2017 when it was ready for print. In August came Bitcoin Cash and segregated witnesses. We covered both. Then in September, came in Bitcoin Core version 0.15, one chapter on that. We also had to test our code on Byzantium, the latest Ethereum release in October 2017. For some time, this book is future proof.

You: Any last words?

Me: Everything that had to go wrong while writing this book went wrong. Code that worked say a month ago stopped working. We had to format our hard disk twice as some blockchain software we installed, stopped other software from running. You see, our code snippets are very brittle so, a small addition to the Bitcoin Networking or Wire Protocol gets our networking code crashing like a ton of bricks. In the year 2015, we had some black hair, in the year 2017, none. This is what Bitcoin, Ethereum and blockchains do to you. Thou have been warned.

You: Your plans for the book.

Me: I plan to write a book on Initial Coin Offerings using Ethereum Smart Contracts. You cannot create a ICO Framework using Bitcoins. Bitcoin will remain either a payment layer or a settlement layer. Blockchain technology adoption is moving at a snail's pace because we need the entire ecosystem to use the same blockchain, one or two major players using blockchain in an industry do not cut it. My belief is that the single biggest use of Smart Contracts is in replacing the Initial Public Offering or a IPO by a ICO. Money is what makes the mare go around. Read the last chapter to understand why. The core idea is to understand Ethereum Smart Contract Security using a business use case.

You: You missed out on important topics like Bitcoin Hardware Wallets.

Me: We bought two Bitcoin Hardware, the original Trezor and the pretender to the throne Ledger Nano S. We should have had a chapter on programming these hardware wallets as they have an API that includes python. If we wrote chapters on everything that was on our wishlist, you would not be reading this book. We broke a dozen deadlines, so we had to draw a line in sand on when to say No.

You: There is no me and you, it's you, writing for me.

Me: Yes, it took you a long time to figure this out J

This is a summary of what every chapter contains

Chapter01. This chapter sets the tone of the book. Using Python code, we understand some of the members of the Bitcoin Block Header using the existing Bitcoin blocks. We also prove that Bitcoin has an arguable Christian influence.

Chapter02. A Bitcoin block mainly stores transactions. Here, we gradually move forward to explain how a Bitcoin transaction is structured into inputs and outputs. A Bitcoin block is mainly made up of transactions.

Chapter03. In this chapter, we have our first encounter with the cryptographic and mathematical constructs used in Bitcoin, the Merkle Tree or Hash. In place of the SHA-256 hash, we calculate the individual hash vales of each transaction and then use Merkle to return a single hash representing all the transactions in a block.

Chapter04. This chapter explains how a Bitcoin address is created along with a checksum. We use the standard library functions and then write our own code to generate a Bitcoin address.

Chapter05. An entire chapter is dedicated to creating vanity Bitcoin address which can be viewed on the Bitcoin blockchain for eternity.

Chapter06. The details of two block header fields, nonce and difficulty which were left unexplained earlier are elaborated here. These fields are used by miners to mine a block. The difficulty field finds out if the miners have successfully mined a block.

Chapter07. The highlight of this chapter is SQL. Using SQL, we store Bitcoin transactions in a Sql database called PostgreSQL. Plus, we verify some basic Bitcoin facts like does the difficulty change every 2016 blocks and more.

Chapter08. The chapter focuses on understanding the role of inputs and outputs in a Bitcoin transaction. Our first gentle introduction to scripting and signatures.

Chapter09. The blockchain contains more information besides the Bitcoin transactions. It contains emails, pdf files, gif files etc. It still is a mystery as to how all this data got into a pristine blockchain.

Chapter10. Signing a Bitcoin transaction is what separates the men from the boys. The chapter reveals the fact that a real Bitcoin transaction with multiple inputs is signed by a certain private key P. A public key generated by the private key P confirms later that this private key P signed the transaction without ever having any access to P.

Chapter11. This chapter is where we put our all our knowledge to practice. We create a raw transaction and send it to a miner to mine. The catch here is that we are not using code written by the Bitcoin Core developers.

Chapter12. This is the smallest chapter in our book. It focuses on the three types of networks that the Bitcoin server can operate on, mainnet, testnet and regtest.

Chapter13. This chapter exposes how we can now legally place any data of our choice in a blockchain. A new script opcode called OP_RETURN is introduced for this purpose.

Chapter14. Bitcoin addresses starting with 3 can get very complex. These Bitcoin addresses are called multi-sig and need more than one private key to sign a transaction if they are to be unlocked and stored in the outputs.

Chapter 15. This chapter and the next chapter explain the Bitcoin networking protocol which is used to download transactions and blocks from miners. Miners, in general, are very pretentious. They will not send you any transaction or blocks unless you talk to them in a certain way. The chapter demonstrates the methodology and the communication protocol between the Bitcoin miners using some shortcuts that give you grief.

Chapter 16. The networking saga continues. The program is enhanced to handle any data sent by a miner, including a 1MB block of transaction data. Further, in the chapter, a series of raw transaction bytes are sent across to a miner without using the sendtransaction method.

Chapter17. The program in this chapter replicates or overrides one of the standard cryptographic library functions. We write a function that creates a sha0/1 hash value instead of using the standard library function. We then check the output with the library function to ensure that the calculated hash value is the same.

Chapter18. This chapter gives an insight into the technique used when computing both the SHA-256 and RIPEMD-160 hash. None of these hashes contain code we are proud of or happy with, especially the RIPEMD-160 hash. We are not cryptographers.

Chapter 19. A public key is just another point on the Elliptic curve. With this knowledge, we sign a transaction using a private key and then verify a signature with only the public key. In conclusion, a random number comes handy when you do not want the world to know your private key.

Chapter 20. This chapter signals the end of our cryptography journey. Here we calculate a public key and signature by using code that does not rely on Sage or any Python library.

Chapter 21. A milestone reached. We write a program to create all the bytes for a transaction and send it over. What's more, our code also signs the raw transaction before putting it on wire.

Chapter 22. This chapter is a step forward. The program now has only our code replacing the standard library functions to create a transaction. Using this approach, we can create every byte of the transaction to be sent to a miner.

Chapter 23. The chapter marks our journey into the undocumented internals of the Bitcoin world. An in-depth study of the index folder where every transaction is stored and the dat files where every block resides. We check over 197 million key-value pairs for consistency.

Chapter 24. The heart and soul of the Bitcoin blockchain is the UTXO set or those transaction hashes that have at least one unspent output. Simple rule: No UTXO set, no blockchain. This is the only practical approach for Bitcoin to keep track of those transaction outputs that are unspent, in a very efficient way. We do the newer UTXO set in another chapter.

Chapter 25. All transactions related to the newly created or existing Bitcoin addresses are stored in a Bitcoin Wallet. We get a thorough understanding of all different keys present in the Berkley DB database wallet.dat.

Chapter 26. After the blk files, the rev or undo files take up the maximum amount of hard disk space. The UTXO set deals with outputs whereas the undo blocks store inputs in a highly-compressed format. This chapter validates the data in each undo block.

Chapter 27. The chapter looks at the two eluded files, banlist.dat and peers.dat, found in the Bitcoin folder. The banlist

file works as advertised. It is the peers file that decides using cryptography, which peers we can connect to. Once connected, we can safely download the blockchain.

Chapter 28. This chapter focusses on miners and blocks. The initial part is focused on fields like chainwork and mediantime which are not stored on disk. Then, we determine who mined a block. We end by scanning through 200 million transactions.

Chapter 29. We pay miners to carry our Bitcoin transactions. Someone must tell our wallet how much to pay a miner. This estimation of fees or change is where the `fee_estimates.dat` file comes in.

Chapter 30. A chapter we wear on our sleeves. The original Bitcoin source code is compiled and built but with a small twist. The blk file sizes are reduced to 512 KB, instead of 128MB and more. This chapter is what is responsible for this book being written.

Chapter 31. We enter the la-la lands of writing Bitcoin tests which can be written in C++ or Python. Our choice will always be the low-level C++ tests. This chapter carries out a test that checks if a certain transaction is well-formed.

Chapter 32. A few chapters are dedicated to Ethereum, the best known blockchain after Bitcoin. Ethereum is also called the world computer as it runs programs written by others on computers all over the world. The biggest drawback with Bitcoin is that it has a limited programming vocabulary.

Chapter 33. Ethereum like Bitcoin stores key-value pairs in a leveldb database but it does not use blk or rev files. In the same way, Ethereum account balances are also stored in this giant state machine that is made up of only key value pairs. As of October 2107, there are over 300 million such key value pairs. This chapter gives an understanding into the different key value pairs.

Chapter 34. This chapter looks at the differences between the UTXO data set and the Ethereum state machine. A Bitcoin UTXO set stores no account balances, which is not true for Ethereum. Then again, staying true to our style, we have attempted to display the balance of our Ethereum address using some very unstable code.

Chapter 35. Bitcoin Cash a new hard fork, actually has blocks whose sizes are close to 8MB. The heart of this chapter is segregated witnesses, something that will enable the settlement layer within the Bitcoin blockchain. We also throw in a few words of wisdom on segwit2x and why in our view it will not be bigger than Bitcoin Core.

Chapter 36. Bitcoin Core 0.15 released in September 2017. It brings in lots of goodies to the table, mostly unseen. Thankfully the UTXO set was rewritten to make it simpler but faster. We also cover smaller topics like replay attacks, what is the actual size of a block, the structure of a mempool file and how Bitcoin creates a caching hash and finally pruning the blockchain.

Chapter 37. This is a revision of different error checks that go into confirming a Bitcoin transaction. We learn a few things about dust and how does one calculate it. We also learn that the size of the signature and public key in the inputs cannot exceed 1650 bytes. Finally, we compute the sigops or signature opcodes present in a transaction.

Chapter 38. Here we start with our version of the future of ICO's. Our focus is on Initial Coin Offerings someday replacing a Initial Public Offering or IPO. An ICO allows us to create or own digital token or currency and allows the world to participate in what we do. We have an Ethereum Improvement Proposal called EIP-20 or a Ethereum Request for Comments ERC-20 that standardises what a digital token should look like. We end with reverse engineering Solidity Bytecodes and Ethereum Smart Contract Security.

Chapter 39. In our journey, all the way through this book, we tried defining a Bitcoin and a blockchain. We may be able to hammer out an agreement on the definition of a Bitcoin, but not on blockchain, not in the present nor in the future. So, everyone has the right to say their product is based on blockchain technologies. It comes as no surprise that there is an umbrella of blockchains in this realm.

Chapter 40. Artificial Intelligence is the biggest blockbuster technology that will change the solar system. Blockchain comes in a close second. We will actually build some basic AI models using a python library called Keras. We found no use case where blockchain and AI technologies could work in sync. Our biggest grouse with AI is that it is not a black box but a black hole which nobody understands.

Table of Contents

Chapter 01: Basics of the Bitcoin Block Header	1
Chapter 02: Transactions - Basics	15
Chapter 03: Computing the Merkle Hash	28
Chapter 04: Bitcoin Addresses	39
Chapter 05: Vanity Bitcoin Addresses	58
Chapter 06: Difficulty and Nonce	75
Chapter 07: Storing Bitcoin Transactions using SQL	93
Chapter 08: Transactions - Inputs and Outputs	127
Chapter 09: Hiding Data in the blockchain	144
Chapter 10: Signing Transactions	155
Chapter 11: Roll your own transaction	165
Chapter 12: Client and Server	194
Chapter 13: Notaries and OP_RETURN	203
Chapter 14: Pay to Script Hash or Multi-Sig Bitcoin addresses	212
Chapter 15: Basic Networking	238
Chapter 16: More Networking	265
Chapter 17: Hashes SHA0 and SHA1	293
Chapter 18: Hashes - Sha-256 and RipeMD-160	321
Chapter 19: ECC with Sage - Part 1	339
Chapter 20: ECC with Sage Part 2	364
Chapter 21: Sending our own transaction	393
Chapter 22: Sending one transaction without using library functions	425
Chapter 23: Index folder	440
Chapter 24: UTXO Dataset	468
Chapter 25: Wallets	524
Chapter 26: Rev/Undo files	560
Chapter 27: peers.dat and banlist.dat	601
Chapter 28: Miners, blocks and more	618
Chapter 29: fee_estimates.dat	649

Chapter 30: Building the Bitcoin Source code	677
Chapter 31: Testing Bitcoin for bugs	687
Chapter 32: Ethereum Solidity	701
Chapter 33: Ethereum leveldb keys and GOLANG	726
Chapter 34: Ethereum Unravelling the State Machine	758
Chapter 35: Bitcoin Cash vs Segwit vs Segwit2x	777
Chapter 36: Bitcoin Core 0.15, UTXO and more	808
Chapter 37: Transactions and Blocks - Error Checks	844
Chapter 38: ICO and Smart Contract Security	859
Chapter 39: What is a Bitcoin and a Blockchain	882
Chapter 40: AI and Blockchain – Never The Twain Shall Meet	887

CHAPTER 01

Basics of the Bitcoin Block Header

Bitcoin is the most successful distributed peer to peer network created by humanity. The combined market capitalization of Bitcoin was over US\$12.5 billion in December 2016, which is by far larger than the market capital of all the other crypto currencies put together. What comes closest is Ethereum (the World Computer), which has a US\$600 million capitalization and at the third position, there is Ripple, which is owned by a corporate entity running at less than US\$250 million.

This book is not intended to bore you with radical questions as to why Bitcoin is the most successful one in the market, instead, it explores the internal workings of the technology behind it. The focus is largely on understanding every byte of the Bitcoin transactions downloaded on your system. Along the way, we also understand all the artifacts that Bitcoin creates on your computer, some well-documented, some only documented in code.

The Bitcoin code base is written only in C++. However, we will write our code in Python initially to understand the basic Bitcoin concepts and then look at C++. The book is filled up with only code. After all, a program speaks louder than a 1000 words or a picture.

So, let's start from the very beginning.

We download the official Bitcoin client called Bitcoin Core from the Bitcoin website, <https://bitcoin.org/en/download>. The version of our client is 0.13.1 but you can download any version you like. However, we advise you to stick to the latest version.

Next, we install the Bitcoin Core using the standard installation process as per the operating system and then run the application. You will be asked only once, to select a folder. In this folder, it will download all the Bitcoin transactions executed to date.

Please keep the default folder name as recommended by the app, /Users/vijaymukhi/Library/Application Support/Bitcoin. The username on my Mac is vijaymukhi, yours will be different obviously. This folder name can also be referred as ~/Library/Application Support/Bitcoin/.

On Windows OS, the folder name is C:\Users\YourUserName\AppData\Roaming\Bitcoin and on Linux, it is ~/.bitcoin/. For the Mac and Linux, they could have had the same folder name but they chose otherwise. Some mysteries, we will never be able to solve. A couple of programs in the forthcoming chapters look at this folder by default, for the Bitcoin database of transactions.

Once the folder is in place, the installation program now begins the download of all the Bitcoin transactions executed in the past. This download could take a couple of hours or a week depending upon your Internet speed as it is about 100Gb and more. You do not have to sit back and wait for the entire download to take place.

This database of transactions stored on your hard disk is also called a distributed ledger, or Bitcoin blockchain, a trust machine etc. Everyone on the Bitcoin network share the same Bitcoin dataset of transactions, thus we all will have an identical copy of it.

In the standard folder which we specify earlier, there is a folder called blocks with around 500++ files having names like blk00000.dat, blk00001.dat and so on and so forth. Each of these files is about 134 MB large. Since it is not very easy to deal with such large files, a minor inconvenience, we use a program called dd, a linux based program which reduces the size of these files as per our bidding. This step is optional.

We open a terminal and run the following command in the blocks folder; /Users/vijaymukhi/Library/Application Support/Bitcoin/blocks (on the Mac/linux only)

```
>dd if=blk00000.dat of=vijay1.dat bs=1000 count=1
```

Every command in Unix/Linux has its own syntax and so does dd. There is no consistency in the command syntax, one of the vagaries of the Unix/Linux command line interface.

The options used with the dd command:

- if stands for input file that we want to truncate
- blk00000.dat contains the Bitcoin distributed ledger
- of stands for output file
- bs is the number of bytes to copy (technically it is the block size but it works)
- count is the number of times we want to copy bs bytes.

So, the number of bytes/blocks copied will be count * bs (1 * 1000) as count is 1 and bs is 1000. As a result, the output file, vijay1.dat is created which is a 1000 bytes large and it represents the initial Bitcoin transactions.

Now, create a new folder and copy vijay1.dat there. All the code and programs, henceforward, will be written in this folder.

To open the file vijay1.dat, we use a hex editor called Synalyze It! Pro, you can use any hex editor available to you. The first 4 bytes of the file are the bytes : F9BEB4D9

These initial bytes are the file's magic number. Bitcoin transactions are grouped together into blocks and each block starts with the magic number F9BEB4D9. In java, the .class file has the magic number of CAFEBABE. Indeed, the Java developers had some sense of humor unlike the ones in the Bitcoin world, who fall in the category of the bankers as they deal with money, old and staid. Nothing creative in the choice of their magic number!!

The next 4 bytes following the magic number is the block size field. The actual size of the block containing transactions can be determined using this field. The size of a block in Bitcoin can change but it cannot exceed 1MB. As an aside, this is what the 1MB civil war in Bitcoin is all about; the fact that the block size cannot exceed 1MB. There have been ugly and ongoing debates on the cap but no resolution so far L

These 4 bytes reveal the size of the transaction data. In our case, it shows the hex values of 1D 01 00 00.

The first byte, 1D, in decimal is 29. To convert a hexadecimal value to decimal, we multiply the first digit 1 by 16 and add the next digit, as in D, which is 13 in decimal. The sum value of $16 + 13 = 29$. Thus, 1D in hex is 29 in decimal. The second byte has a value of 01 and it is multiplied by 256. The value of 256 is because 2^8 is 256. Similarly, the value in the third byte, 0 is multiplied by 2^{16} . The fourth byte value, 0 will be multiplied by 2^{24} .

The sum or total is a very big number. We stop our calculations at the first 2 bytes since the next two bytes are both 0's. All in all, the value of the integer stored in these next four bytes is $256 + 29 = 285$. The size of the block is 285 bytes.

The next 285 bytes will hold all the transaction data of the block.

In a .dat file, the blocks are placed in a sequential manner, one after another. So, the next block will start at byte position $285 + 8 = 293$ from the start of the file. The first 4 bytes for the magic number plus the next 4 bytes for the block size and then the block data (285).

Every hex editor gives an option to jump to a certain part of the file. Using the option in the editor, we move to location 293. Voila, we once again see the magic number F9BEB4D9, the start of a fresh block.

Now it will take eons to understand Bitcoin technology in this manner, so let's use Python code to understand the world of Bitcoins better.

Most of our programs are short and sweet and to the point. And, all our variable names are also small in place of bigger more meaningful names, to limit the program size.

Reading the First Byte of the First Block

We have learnt so far that the first byte of the magic number is the hex value, 0xf9. So, we write a program to display this byte.

We assume you have a fair knowledge of the Python programming language. So, let's start. Using any text editor, create a new file with .py extension in the folder having vijay1.dat.

```
ch0101.py
f = open("vijay1.dat")
a = f.read(1)
print a
print type(a)

>python ch0101.py
```

```
Output
?
<type 'str'>
```

In the program, we open the file vijay1.dat using the Python function, open. The file object returned by the function is saved in the variable/object called f.

Then comes the read function, which is a part of the file object. This function takes a single parameter, which is a count or the number of bytes to read from the file. We want to read a single byte so 1. It is the first byte of the file vijay1.dat. The returned value is stored in variable a. The print command in python displays the value in a with its datatype.

The read function returns a string so the number 65 or 0x41 will show A on the screen with its datatype. Most of the numbers above 128 will be displayed as junk or unprintable characters. A file on disk only contains numbers between 0 to 255.

So, lets modify the program to get some readable characters.

Displaying the first byte of the magic number in hex format

```
ch0102.py
f = open("vijay1.dat")
a = f.read(1)
print "%x" % ord(a)
```

```
Output
f9
```

Now things look much better. We see the first byte as f9.

The read function returns a string, as always. We then use the ord function which takes the first byte of the string and converts it to a number. The file on disk has the same number. Finally, using the modifier %x of the print command, we display this number in hex format.

Reading and Displaying the Magic Number

```
ch0103.py
f = open("vijay1.dat")
a = f.read(4)
print a
print a.encode('hex')
print "%x%x%x%x" % (ord(a[0]), ord(a[1]), ord(a[2]), ord(a[3]))
```

Output

```
????
f9beb4d9
f9beb4d9
```

The magic number is stored in first four bytes of the file on the disk. The program reads the first four bytes from disk and displays it using the print function. The first print function shows the four ?'s or four unprintable characters as it shows a string.

For the second print command, we use the encode function from the string class which simply converts every number in the string to an equivalent hex digit. Since we require a hex value, the encode function is supplied the parameter 'hex'.

Alternatively, a simple call to the ord function on every byte of the string object can be used, it gives the same result. The array notation accesses the string members one byte at a time.

Writing Our Own String Encode Function

```
ch0104.py
def encode(s):
    str = ""
    i = 0
    while ( i < len(s) ):
        d = "%02x" % ord(s[i])
        str = str + d
        i = i + 1
    return str

f = open("vijay1.dat")
a = f.read(4)
print a.encode('hex')
print encode(a)
```

Output

```
f9beb4d9
f9beb4d9
```

This program outlines our approach in this book. Initially, we use an in-built function in Python or maybe an external

library and explain it. Then we write our own function to perform the same task. The above program mimics the encode function.

The len function determines the size of the string given to the encode function, since the size keeps changing. Then, a while loop is put in place to access every character in the string. The slice or the array operator [] comes in handy. As before, the ord function converts every string character to an equivalent value in hex and the %x modifier converts a decimal number into a hex value. This value is stored in a string form into a temporary variable d. The string str is concatenated with the string d throughout the loop. At the end of it all, the variable str stores the final encoded string.

Computing the Size of the Block

```
ch0105.py
f = open("vijay1.dat")
magic = f.read(4)
i = ord(f.read(1))
j = ord(f.read(1))
print i , j , i*1 + j * 256
```

Output
29 1 285

The variable magic stores the magic number. And the next 2 bytes are read in variable i and j. We know the four bytes following the magic number is the size of the Bitcoin block. The size of the first block is 285 bytes, as shown by the hex editor earlier.

In this program, the variable i and j hold these values, which are 29 and 1. The value in the first byte, i is multiplied by 1 and the value in the second byte, j with 256. The total of these values i.e. 285 is then displayed on the screen. If the next two bytes are further extracted from the file, then the third byte is multiplied by 65536 or 2^{16} and the last or fourth byte by 2^{24} . The total of value in these bytes is the size of the block.

This, no doubt, is a very tedious way of reading an integer from a binary file. There is a better approach.

The Use of the Unpack Function to Read an Integer

```
ch0106.py
import struct
f = open("vijay1.dat")
f.read(4)
size = f.read(4)
size = struct.unpack("I" , size)
print size
print type(size)
print size[0]
```

Output
(285,)
<type 'tuple'>
285

The above program first reads the magic number without saving it and then reads the next four bytes into an aptly named string variable, size.

The Python module struct has a function called unpack which is used to extract binary values in a string. The first parameter to this function is a data type and the second is the string that contains the actual binary data. It returns a Python data type called a tuple.

A tuple stores multiple values in () brackets, each of them separated by a comma. The reason being that every string can have data with multiple data types, and they are stored one after the other.

The value l given to the unpack function stands for an integer. The unpack function reads the four bytes in the given string, size and gives out an equivalent integer value. The function does the multiplication work internally. We see the round brackets and the comma following 285. It simply proves that the variable size, earlier a string, is of data type tuple now.

As we are interested in only the first value present in the tuple, we use size[0] operator to access its integer value.

Reading the First Two Fields of the Block

```
ch0107.py
import struct
f = open("vijay1.dat")
magic = f.read(4)
size = f.read(4)
size = struct.unpack("l", size)[0]
print "Magic Number is %s" % magic.encode('hex')
print "Block Size is %d" % size
```

Output

```
Magic Number is f9beb4d9
Block Size is 285
```

This program places everything under one roof and displays a neat and tidy output.

After displaying the encoded magic number, we read the next four bytes and unpack them. The variable size now a tuple, stores the integer value 285 present in the first object. The value is then displayed on the screen.

Reading the First Two Fields of the First Three Blocks

```
ch0108.py
import struct
f = open("vijay1.dat")
for i in range (0,3):
    print "Block Number %d" % i
    magic = f.read(4)
    size = f.read(4)
    size = struct.unpack("l", size)[0]
    print "\tMagic Number is %s" % magic.encode('hex')
    print "\tBlock Size following is %d" % size
    f.seek(size , 1)
```

Output

```
Block Number 0
    Magic Number is f9beb4d9
```

```

    Block Size following is 285
Block Number 1
    Magic Number is f9beb4d9
    Block Size following is 215
Block Number 2
    Magic Number is f9beb4d9
    Block Size following is 215

```

This program displays the magic number and the block size of the first 3 blocks of the Bitcoin blockchain.

The for statement repeats the code written in the earlier program, multiple times. The range function returns the value 0,1,2 as the parameters are 0,3. So, the code is executed 3 times.

The newly added seek function in a file object jumps a certain number of bytes from the defined location. The first parameter of the seek function is the number of bytes to be jumped over and the second parameter is a relative position. The value in the second place can be 0 which is the beginning of the file, 1 is relative or current position and 2 is the end of the file.

Every block starts with a magic number, followed by size and then data. After reading the size field, the seek functions jumps the size bytes and reaches the start of the next block. And the same procedure is repeated. In the Bitcoin protocol, there is no concept of record number in the physical block, but there is a virtual block number.

After the first 8 bytes, comes the data section of the block. This section starts with an 80-byte block header. Let us now display the members of this header one by one.

Reading the First Two Fields of the Bitcoin Block Header

```

ch0109.py
import struct
f = open("vijay1.dat")
for i in range (0,3):
    print "Block Number %d" % i
    magic = f.read(4)
    size = f.read(4)
    size = struct.unpack("l", size)[0]
    print "Magic Number is %s" % magic.encode('hex')
    print "Block Size following is %d" % size
    header = f.read(80)
    (ver , prevhash ) = struct.unpack("l32s" , header[0:36])
    print "\tVersion Number is %d" % ver
    print "\tPrevious Block Hash is %s" % prevhash.encode('hex')
    f.seek(size - 80, 1)

```

Output

```

Block Number 0
Magic Number is f9beb4d9
Block Size following is 285
    Version Number is 1
    Previous Block Hash is
0000000000000000000000000000000000000000000000000000000000000000

```

```
Block Number 1
Magic Number is f9beb4d9
Block Size following is 215
    Version Number is 1
    Previous Block Hash is
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
Block Number 2
Magic Number is f9beb4d9
Block Size following is 215
    Version Number is 1
    Previous Block Hash is
4860eb18bf1b1620e37e9490fc8a427514416fd75159ab86688e9a83000000000
```

After the four bytes of magic number and four bytes of the block size, there is an 80 byte data structure called Bitcoin block header. It has six fields.

The first member of the block header is the version number field of type integer (4 bytes). It has a value of 1 as of writing this book. The version number field is required because the Bitcoin protocol will change over a period of time. Plus, it allows us to identify and maintain compatibility between the older and newer versions.

Everyone in the Bitcoin ecosystem is expected to modify their code to take advantage of the newer features in the updates. Though, it is not a trivial task. At times, problems do surface when some clients upgrade and some do not. That's when the compatibility with old client code comes in.

The second member after the version field is a 32-byte hash field. We will explain the concept of hash in some time.

In the program, the read function reads the entire block header into a string called header.

Then, we use the unpack function to unpack only two out of the six fields, the version number and the hash value.

Since the version number is an integer, we use I in the unpack function to extract the value. At present and for a long time, the value will remain as 1.

The hash value picked up from the binary file is a 32-byte number. It has unprintable characters, so the function encode is used to convert these binary numbers into printable hex bytes. A text file has legible characters so the encode function is of no use there.

Back again, the unpack function is given I to read the first 4 bytes and then 32s for a string of 32 bytes. The second parameter is the header field with the slice operator as we need only the first 36 bytes.

After displaying the relevant data, we proceed to the next block. The seek function with parameters (size – 80) from the current position performs this task. This way, we display the first 2 fields from the block header of the first 3 blocks.

Reading the Entire Bitcoin Block Header

```
ch0110.py
import struct
import time
f = open("vijay1.dat")
for i in range (0,3):
    print "Block Number %d" % i
    magic = f.read(4)
    size = f.read(4)
```



```

size = struct.unpack("l", size)[0]
print "Magic Number is %s" % magic.encode('hex')
print "Block Size following is %d" % size
header = f.read(80)
(ver, prevhash, merklehash, time1, diff, nonce) = struct.unpack("l32s32slll", header)
print "\tVersion Number is %d" % ver
print "\tPrevious Block Hash is %s" % prevhash.encode('hex')
print "\tMerkle Hash is %s" % merklehash.encode("hex")
print "\tDate Time %d" % time1
print "\t%s" % time.ctime(time1)
print "\tDifficulty %x" % diff
print "\tNonce %d" % nonce
f.seek(size - 80, 1)

```

Output

Block Number 0

Magic Number is f9beb4d9

Block Size following is 285

Version Number is 1

Previous Block Hash is

00

Merkle Hash is

3ba3edfd7a7b12b27ac72c3e67768f617fc81bc3888a51323a9fb8aa4b1e5e4a

DateTime 1231006505

03-January-2009 23:45:05

Difficulty 1d00ffff

Nonce 2083236893

Block Number 1

Magic Number is f9beb4d9

Block Size following is 215

Version Number is 1

Previous Block Hash is

6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000

Merkle Hash is

982051fd1e4ba744bbbe680e1fee14677ba1a3c3540bf7b1cdb606e857233e0e

DateTime 1231469665

09-January-2009 08:24:25

Difficulty 1d00ffff

Nonce 2573394689

Block Number 2

Magic Number is f9beb4d9

Block Size following is 215

Version Number is 1

Previous Block Hash is

4860eb18bf1b1620e37e9490fc8a427514416fd75159ab86688e9a8300000000

Merkle Hash is

d5fdcc541e25de1c7a5addedf24858b8bb665c9f36ef744ee42c316022c90f9b

```
DateTime 1231469744  
09-January-2009 08:25:44  
Difficulty 1d00ffff  
Nonce 1639830024
```

This program displays the entire block header.

The field following the hash field in the block header is another hash, again 32 bytes large. This is the Merkle hash which we will discuss in a short while. We delay explaining a few concepts because it may lead to massive headaches due to information overload.

The 4 bytes after the Merkle hash refers to time, in seconds. The value is the duration or the time difference in seconds from the block creation time to the 1st of January 1970, like the Unix Operating System. The function ctime from the time module converts the seconds into a proper date-time format. The program displays the time in seconds as well as in date-time format.

Our program output shows that the first block or the Genesis block was created on the 3rd of January 2009. Then, it looks like the Bitcoin God Satoshi takes a break. He creates the second Bitcoin block 6 days later, on the 9th of January 2009. Coincidentally, the first block is called the Genesis block in nearly all crypto currencies.

The second last field is called 'difficulty' and talks majorly about the calculations to be performed by the miners on the block. The last field is called 'nonce' which decides on how many times we need to calculate the hash value of this header. Both the fields hold numerical values. These two fields go hand in glove and are explained in greater detail in the subsequent chapters.

Before we proceed further, let's understand hash in simple terms.

A hash value is simply a number created using an algorithm to represent some bytes. It is widely used to avoid any tampering to the data while transmitting it on the network.

A simple example would be, let's say sending the characters ABC to a friend. When this data travels, it goes in the ASCII format, so 65 66 67 goes on the wire. There is now a possibility that the data can get corrupted or tampered before reaching the destination. However, there is no way to check if the data has reached unchanged, so we hash it. One of the simplest hash algorithm is to add the ascii values of these 3 numbers, i.e. $65 + 66 + 67 = 198$, and email/phone/text it, basically using a different channel.

On receiving the data, our friend can simply run the same algorithm i.e. add the ASCII values and check the total, 198. If the total matches then the data is the same.

The glitch with this approach is that the hacker can rearrange the numbers, i.e. ACB, or CBA, and the hash value will be the same.

So, our next attempt is to add a weight to every number, depending upon its position in the file. The hash is now calculated as $(65 * 1) + (66 * 2) + (67 * 3)$ which is 398. With this approach, the rearranging of numbers will result in a different hash.

Nevertheless, this method is still not trustworthy as many combinations of data give a hash of 398. The need of the hour is a way of representing the bytes/text as a hash value and in a manner that a change in one byte alters the hash beyond recognition.

The Bitcoin world use the hash algorithm called SHA-256 (Secure Hash Algorithm), where 256 is the size of the hash as in 256 bits or 32 bytes.

This value holds great significance which we will unravel in the later chapters.

How the Bitcoin Blocks are Internally Linked to Each Other

```

ch0111.py
import struct
import hashlib
f = open("vijay1.dat")
for i in range (0,3):
    print "Block Number %d" % i
    magic = f.read(4)
    size = f.read(4)
    size = struct.unpack("l" , size)[0]
    header = f.read(80)
    (ver , prevhash , merklehash , time , diff , nonce ) = struct.unpack("I32s32sIII" , header)
    print "\tPrevious Block Hash is %s" % prevhash.encode('hex')
    shahash = hashlib.sha256(header).digest()
    shahashf = hashlib.sha256(shahash).digest()
    print "\tCurrent Block Hash is %s" % shahashf.encode('hex')
    f.seek(size - 80, 1)

```

Output

```

Block Number 0
    Previous Block Hash is
0000000000000000000000000000000000000000000000000000000000000000
    Current Block Hash is
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
Block Number 1
    Previous Block Hash is
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
    Current Block Hash is
4860eb18bf1b1620e37e9490fc8a427514416fd75159ab86688e9a8300000000
Block Number 2
    Previous Block Hash is
4860eb18bf1b1620e37e9490fc8a427514416fd75159ab86688e9a8300000000
    Current Block Hash is
bddd99ccfda39da1b108ce1a5d70038d0a967bacb68b6b63065f626a00000000

```

Finally, the chickens have come home to roost.

Bitcoin transactions generally get confirmed in about 10 minutes. However, at present, the wait is much longer. These transactions are added in the blocks of a distributed ledger or the blockchain.

A trillion-dollar question here is that, since the blocks holding the bitcoins are stored in a simple .dat file, rather than a database, are all the blocks organized in a sequential manner? How is the chronological order and the links maintained in the blockchain? Though, the timestamp field serves this purpose to some extent but, it is not used.

The Bitcoin world has an elegant solution to this problem. There is a field called previous block hash in the block header, which stores the SHA hash of the previous block. This previous block hash field stores the hash of the 80-byte block header only. However, through our code, we also learnt that it represents the hash of the entire block ultimately, one of the things we cover in the later chapters in the book. It must be noted that the hash of the current block is not stored anywhere in the block.

The first block, Block-0 is a hard-coded block in the Bitcoin source code and it has a value of 0 in the previous hash block field, obviously! This will be the only block in the blockchain with zeroes in the previous hash block field.

As mentioned earlier, the SHA-256 hash algorithm is used to calculate the hash value of the 80 bytes in the block header of the genesis block. Understanding the design of the SHA-256 algorithm would take us a lifetime and we believe there are less than 100 people in the world who know the formula. On the other hand, its usage is very simple. Functions like sha256 and digest from the hashlib library are used for this purpose.

The point to be noted always is that the hash value of the block header is not stored anywhere in the current block. Its value is assigned to the previous block hash field in the next block. Which means that the previous block hash field of block 1 holds the computed hash value of the block 0 header. In the same way, the previous hash field of the second block is set to the computed hash value of the block header of block 1 and so on and so forth.

In this manner, a chronological order to the blocks is maintained in the blockchain. Every block in the blockchain maintains a link to the previous block and so on. We believe this approach is by far better than record numbers for various reasons.

Since a block hash uniquely represents block header data, any changes made to this data will inadvertently change the hash value and thus break the link.

Let's look at the program code.

The module or library hashlib has code which computes different types of hashes. So, we import this library. The sha256 function in the library computes the sha256 hash. Our objective is to get a hash value of the header and save it in the previous block hash field of the next block.

First, the contents of the 80-byte block header are read in the variable called header. It is then unpacked into the six members individually in its own datatype. The hex values stored in prevhashblock is then displayed.

Now comes the interesting part.

The sha256 function from the hashlib library is given the 80-byte header and the digest function thereafter computes its hash value, which is stored in the variable shahash. This value is again given to the functions, sha256 and digest for a second hash computation. It is done for security reasons, though we have no proof of it. The double hash value is stored in a variable shahashf which is then displayed in hex format.

The output shows that the Previous Block Hash of block 1 is the same as the computed hash value of the header of block 0. Similarly, the previous block hash value of block 2 stores the hash value of the block 1 header.

One of the many reasons the Economist called the Bitcoin Blockchain a Trust machine is because every block is linked or chained to the previous block and not a single byte can be changed after the block is added to the blockchain.

All the Bitcoin transactions are placed after the 80-byte header. One block is used to store multiple transactions. There is an average wait time of 10 minutes to collect all the transactions that have taken place and then put in a block. The Merkle hash is a hash value of all the transactions placed in the block.

The next program displays the Merkle hash value of transactions in the first 3 blocks.

Calculating the Merkle Hash

```
ch0112.py
import struct
import hashlib
f = open("vijay1.dat")
for i in range (0,3):
```

```

print "Block Number %d" % i
magic = f.read(4)
size = f.read(4)
size = struct.unpack("l" , size)[0]
header = f.read(80)
(ver , prevhash , merklehash , time , diff , nonce ) = struct.unpack("I32s32sIII" , header)
trans = ord(f.read(1))
print "Number of transactions are %d" % trans
trandata = f.read(size - 80 - 1)
shahash = hashlib.sha256(trandata).digest()
shahashf = hashlib.sha256(shahash).digest()
print "Merkle Hash in Block Header is %s" % merklehash.encode('hex')
print "Calculated Transaction Hash is %s" % shahashf.encode('hex')

```

Output

Block Number 0

Number of transactions are 1

Merkle Hash in Block Header is

3ba3edfd7a7b12b27ac72c3e67768f617fc81bc3888a51323a9fb8aa4b1e5e4a

Calculated Transaction Hash is

3ba3edfd7a7b12b27ac72c3e67768f617fc81bc3888a51323a9fb8aa4b1e5e4a

Block Number 1

Number of transactions are 1

Merkle Hash in Block Header is

982051fd1e4ba744bbbe680e1fee14677ba1a3c3540bf7b1cdb606e857233e0e

Calculated Transaction Hash is

982051fd1e4ba744bbbe680e1fee14677ba1a3c3540bf7b1cdb606e857233e0e

Block Number 2

Number of transactions are 1

Merkle Hash in Block Header is

d5fdcc541e25de1c7a5addedf24858b8bb665c9f36ef744ee42c316022c90f9b

Calculated Transaction Hash is

d5fdcc541e25de1c7a5addedf24858b8bb665c9f36ef744ee42c316022c90f9b

The program retrieves the Merkle hash value of the transactions from the header and then displays it. In addition to it, we have used functions from the hashlib library to compute the hash value of the transaction data and displayed it as well. Both values are the same.

As always, the block header is read into a set of variables. This brings us to the start of the transaction data. The first field in this section is a count of the number of transactions in the block. This value is placed in only one byte and not four bytes. We read this number and then use the ord function to convert it into a number. Fortunately for us, the first 169 blocks are made up of just one transaction.

The size variable in our program has the block size which is the total size of the block. So basically, it is a sum of three entities, namely the 80 bytes of the block header followed by one byte for the count of transactions and then the size of all the transaction data.

The read function reads all the bytes after 81 bytes and stores them in a string variable, trandata. So, trandata now holds all the transactions in the block minus the transaction count byte.

As before, we calculate a hash value of all the transactions using the sha256 digest functions, in our case it's just 1 transaction.

The newly minted hash value matches the value stored in the Merkle root field. This only proves that the Merkle hash is the hash of all the transactions in the block.

We will also revisit our definition of the Merkle root hash in one of the chapters again for more clarity.

It can thus be concluded that the block header hash represents the entire block even though it is only a hash of 80 bytes of data. If we change one bit in the transactions data, the Merkle hash changes thus changing the block hash. This one hash value is the hash of all the transactions in the block.

The time field in the block header is constant for the current block but changes for every block. It increments every 10 minutes on an average; so, the next block should have time difference of 10 mins approximately.

To sum up again, every block starts with a four-byte constant magic number followed by four bytes for the block size field, then the 80-byte block header, one byte for the transaction count and finally transaction data.

The thumb rule is that once a transaction is buried at least 7 blocks deep, that transaction is golden and final. It is immutable. Any change to the bits or bytes in a transaction changes the block header hash value. The chain is broken as the link then cannot be established.

Magic no	Block size field	Block Data : Block header	Block data: Number of transactions	Block data : Transaction data
F9BEB4D9	Number < 1MB	80 bytes		

Block data : Block header

Version number	Hash field	Merkle Hash	Time	Difficulty	Nonce
1	32-byte value	32-byte value	Integer	Integer	Integer

CHAPTER 02

Transactions - Basics

First a confession! To simplify matters, in the previous chapter we said there was only 1 byte used to store the count value for the number of transactions in a block. So, in effect, we were casting in stone that the maximum number of transaction per block will be only 256. A sure recipe for disaster. In real life applications, the block may either have one or a billion small transactions.

Normally, the number of transactions in a block does not exceed 255, so assigning 4 bytes for the count field is wasting disk and memory space. The only way out is to make it flexible. A variable number of bytes is used to store an integer, but this is at the cost of making the program code more complex.

The Bitcoin approach is presented in the program given below.

Displaying the Number of Transactions in the Block Header

```
ch0201.py
from cfuncs import *
f = open("vijay1.dat", "rb")
for i in range(0, 3):
    print "-----Block Number is %d" % i
    magic = rbytes(f, 4)
    print "%s" % magic.encode('hex')
    size = rint(f)
    header = f.read(80)
    (ver, prevhash, merklehash, time1, diff, nonce) = struct.unpack("I32s32sIII", header[0:80])
    merklehash = merklehash.encode('hex')
    notran = rvarint(f)
    trandata = f.read(size - 80 - 1)
    print size - 80 - 1
    shahashf = chash(trandata)
    shahashf = shahashf.encode('hex_codec')
    print "Root of the Merkle Hash is %s" % merklehash
    print "Current Transaction hash is %s" % shahashf
    print "Number of Transaction are %d" % notran

cfuncs.py
import struct
import hashlib
def rint(f):
    return struct.unpack("I", f.read(4))[0]
```

```
def rvarint(f):
    size = ord(f.read(1))
    if size < 0xfd:
        return size
    if size == 0xfd:
        return struct.unpack('H', f.read(2))[0]
    if size == 0xfe:
        return rint(f)[0]
    if size == 0xff:
        return r8bytes(f)
    return -1

def rhash(f):
    return f.read(32)

def rbytes(f, size):
    return f.read(size)

def r8bytes(f):
    return struct.unpack('Q', f.read(8))[0]

def chash(s):
    temp = hashlib.sha256(s).digest()
    final = hashlib.sha256(temp).digest()
    return final
```

Output

```
-----Block Number is 0
f9beb4d9
Root of the Merkle Hash is
3ba3edfd7a7b12b27ac72c3e67768f617fc81bc3888a51323a9fb8aa4b1e5e4a
Current Transaction hash is
3ba3edfd7a7b12b27ac72c3e67768f617fc81bc3888a51323a9fb8aa4b1e5e4a
Number of Transaction are 1
-----Block Number is 1
f9beb4d9
Root of the Merkle Hash is
982051fd1e4ba744bbbe680e1fee14677ba1a3c3540bf7b1cdb606e857233e0e
Current Transaction hash is
982051fd1e4ba744bbbe680e1fee14677ba1a3c3540bf7b1cdb606e857233e0e
Number of Transaction are 1
-----Block Number is 2
f9beb4d9
Root of the Merkle Hash is
d5fdcc541e25de1c7a5addedf24858b8bb665c9f36ef744ee42c316022c90f9b
Current Transaction hash is
d5fdcc541e25de1c7a5addedf24858b8bb665c9f36ef744ee42c316022c90f9b
Number of Transaction are 1
```


If you look at the output of this program, it has an uncanny resemblance to the one in the previous chapter. But the code that generates this output looks very different and it is professionally written.

The new variation in the program is the introduction of file, `cfuncs.py`. All user-defined functions are placed in this separate python file.

The Python keyword 'from' brings in these functions to our code. There will be less clutter with this approach thus allowing us to focus on the essentials of the code.

The method used by Bitcoin to determine a count for the transactions in a block is as follows:

The first byte of the count field tells us how many bytes are being used to store the number. If the number of transactions are 252, then a single byte is used to store the value of the number.

However, if the value is 253, then the next two bytes store the actual number of transactions in the block. This means that all numbers ranging from 253 to 65535 (2^{16}) i.e. short, are stored in the next two bytes.

If the field has a value of 254, it means that the next 4 bytes i.e. an integer (2^{32}) has the count value.

And in the last case scenario, when the value is 255, the next 8 bytes will hold the 64-bit number.

Before we look at the code, it must be noted that the unpack function with the option of H stands for a short or unpacking of 2 bytes and Q is to unpack the next 8 bytes. We already know that I represents an integer, which is 4 bytes.

An int or integer is frequently read from disk so, to avoid duplication, we place the code in a function, `rint`. This function reads 4 bytes from the disk or the blockchain, discerns an integer value using the handy unpack function and finally returns the new value. Similarly, function `r8bytes` uses the unpack function with a value of Q to unpack the next 8 bytes.

Function `rvarint` puts it all together. In this function, the value present in the first byte is saved in a variable called `size`. This value is then checked against one of the 4 conditions listed above, namely it being less than 0xfd (253), equal to 0xfd (253), equal to 0xfe (254) and then matching 0xff (255). A relevant function is called subsequently based on the criteria it matches to handle individual cases. In case of any discrepancy, we return -1. Though there is no room for error here, we still heed with caution due to our past experiences. All in all, the size can either be stored in 1 byte or 9 bytes and this technique devised by the original coders is very efficient in saving space. Hats off to them.

The other functions in the file are not necessary presently. For e.g., the function `rhash` has one line, where the read function reads 32 bytes from the file. The value returned is in its raw pristine form of unprintable characters and not a string. This is the size of a SHA-256 hash.

Similarly, the function `rbytes` is passed an extra parameter to read a certain number of bytes from the file.

Finally, since the SHA hash is computed multiple times, we place its code in a function, `chash`. This function comprises only two lines, each calculating the SHA hash of the data supplied to it. The double hash approach!!

Displaying Details of all the Transactions Present in the First Three Blocks

```
ch0202.py
from cfuncs import *
def doutput(f , i):
    print "\tOutput Number %d" % i
    bitcoinvalue = r8bytes(f)
    print "\t\tValue of the Bitcoin %d: %.02f" % (bitcoinvalue , bitcoinvalue * 1.0/100000000.00)
    outputscripplen = rvarint(f)
    print "\t\tLength of Output Script %d" % outputscripplen
    scriptpubkey = rbytes(f , outputscripplen).encode('hex')
```

```
print "\t\tOutput Public Key Script %s" % scriptpubkey
def dinput(f , i):
    print "\tInput Number %d" % i
    prevtrhash = rhash(f).encode('hex')
    print "\t\tPrevious Transaction Hash is %s" % prevtrhash
    outputindex = rint(f)
    print "\t\tOutput Index in transaction %x" % outputindex
    inputscriptlen = rvarint(f)
    print "\t\tInput Script Length ScriptSignature %d" % inputscriptlen
    sigpubkey = rbytes(f , inputscriptlen).encode('hex')
    print "\t\tSignature + Public Key %s" % sigpubkey
    seqno = rint(f)
    print "\t\tSequence Number %x" % seqno
f = open("vijay1.dat" , "rb")
for blkno in range (0 , 3):
    print "-----Block Number is %d" % blkno
    magic = rbytes(f, 4)
    print "Magic Number is %s" % magic.encode('hex')
    size = rint(f)
    header = f.read(80)
    notran = rvarint(f)
    print "Number of Transactions are %d" % notran
    for tno in range ( 0 , notran):
        print "Transaction number %d in block %d" % (tno , blkno)
        tver = rint(f)
        print "\tVersion Number of Transaction is %d" % tver
        noinputs = rvarint(f)
        print "\tNumber of inputs are %d" % noinputs
        for i in range ( 0 , noinputs):
            dinput(f , i)
        nooutputs = rvarint(f)
        print "\tNumber of outputs are %d" % nooutputs
        for i in range ( 0 , nooutputs):
            doutput(f , i)
    locktime = rint(f)
    print "\tLockTime on transaction is %x" % locktime
```

Output

```
-----Block Number is 0
Magic Number is f9beb4d9
Number of Transactions are 1
Transaction number 0 in block 0
    Version Number of Transaction is 1
    Number of inputs are 1
    Input Number 0
        Previous Transaction Hash is
```

```
0000000000000000000000000000000000000000000000000000000000000000
    Output Index in transaction ffffffff
    Input Script Length ScriptSignature 77
    Signature + Public Key
04ffff001d0104455468652054696d65732030332f4a616e2f32303039204368616e63656c6c6
f72206f6e206272696e6b206f66207365636f6e64206261696c6f757420666f722062616e6b73
    Sequence Number ffffffff
    Number of outputs are 1
    Output Number 0
    Value of the Bitcoin 5000000000:50.00
    Length of Output Script 67
    Output Public Key Script
4104678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb
649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5fac
    LockTime on transaction is 0
-----Block Number is 1
Magic Number is f9beb4d9
Number of Transactions are 1
Transaction number 0 in block 1
    Version Number of Transaction is 1
    Number of inputs are 1
    Input Number 0
        Previous Transaction Hash is
0000000000000000000000000000000000000000000000000000000000000000
        Output Index in transaction ffffffff
        Input Script Length ScriptSignature 7
        Signature + Public Key 04ffff001d0104
        Sequence Number ffffffff
    Number of outputs are 1
    Output Number 0
        Value of the Bitcoin 5000000000:50.00
        Length of Output Script 67
        Output Public Key Script
410496b538e853519c726a2c91e61ec11600ae1390813a627c66fb8be7947be63c5
2da7589379515d4e0a604f8141781e62294721166bf621e73a82cbf2342c858eeac
    LockTime on transaction is 0
-----Block Number is 2
Magic Number is f9beb4d9
Number of Transactions are 1
Transaction number 0 in block 2
    Version Number of Transaction is 1
    Number of inputs are 1
    Input Number 0
        Previous Transaction Hash is
0000000000000000000000000000000000000000000000000000000000000000
        Output Index in transaction ffffffff
        Input Script Length ScriptSignature 7
```

```
Signature + Public Key 04ffff001d010b
Sequence Number ffffffff
Number of outputs are 1
Output Number 0
Value of the Bitcoin 5000000000:50.00
Length of Output Script 67
Output Public Key Script
41047211a824f55b505228e4c3d5194c1fcfaa15a456abdf37f9b9d97a4040afc07
3dee6c89064984f03385237d92167c13e236446b417ab79a0fcae412ae3316b77ac
LockTime on transaction is 0
```

This program displays the structural details of all transactions present in the first three blocks.

A transaction is made up of only two simple structures, namely, inputs and outputs. An input performs a dual role: determining the output handler, as in who is the source of the money and secondly, the true owner who can claim ownership. Similarly, an output confirms the bitcoin address who is to receive the money in future or who becomes the next input handler. It also decides on the rules of claiming this output in the future.

Bitcoins do not come from thin air, they come as outputs of previous transactions and then become inputs to new transactions. Unless the Bitcoins are not received in the past, we cannot give it to anyone. The input side thus collects/ receives money which will be then distributed as outputs to others.

A block will have multiple transactions, therefore there will be multiple inputs and outputs. Inputs gives us details on the source, where the money originated from; it can be from different sources or outputs.

It may be a bit confusing at this moment, but stay with us. The book is dedicated to understanding inputs and outputs in a simplified way, so towards the end it will be all clear.

Now let's understand the input and output structures through our program

After reading the initial 8 bytes and the block header, we save the number of transactions in the block into a variable `notran`. Our newly minted function, `rvarint` computes this number based on a varying number of bytes. This field, i.e. the number of transactions is obviously not part of the transaction structure.

The for loop will iterate some multiple times depending upon the number transactions we have. For now, we have only 1 transaction in a block so this loop could have been avoided.

The `tno` variable is the loop variable; it displays the current transaction number. For some time, our block's will only have one transaction as these are the first or initial Bitcoin blocks.

The first field of a transaction record is the version number field. For reasons best known to the Bitcoin folks, they store the value in 4 bytes and not 1.

Let's look at the input field. It is assumed that a transaction will have many inputs so we have a for loop in place. The variable `noinputs` follows the same logic of transaction number and gives a count value, as in how many inputs make up the current transaction. For a long time, our transactions will have only one input. We will have only one source or output supplying us the Bitcoins. Frankly speaking, these first set of inputs do not look normal but we will explain that later.

Function `dinput` is used to display the input structure. The first field of the input structure is a hash value.

Let's keep in mind, that the output of a previous transaction is the input of the current transaction. And the output of the current transaction may be the input of the next transaction. An input always refers to a hash of a past/previous transaction.

In all, there are two hashes, a transactions hash and a block header hash.

When the transaction hash is all 0's, as in this case, the hash is marked invalid. It means that we are referring to no valid input in the transaction. This transaction is called a Coinbase transaction and it is a very special case.

A Coinbase transaction has no previous transaction hash to refer to and therefore the transaction hash is all 0's. This is the only input type that creates Bitcoins.

Computers on the Bitcoin network are asked to solve a mathematical puzzle or participate in a Bitcoin game. This puzzle is simply a way of buying or wasting time basically wasting electricity and adding to global warming. It does not add to the world's knowledge or find a cure for cancer. The computer that solves this puzzle is rewarded. Initially the reward was 50 Bitcoins, then it became 25 Bitcoins and now it is 12.5 Bitcoins. This is the only known way of creating fresh Bitcoins.

All computers running the Bitcoin Core software, now called miners, can participate in this 'Bitcoin game' of solving irrelevant puzzles. A Coinbase transaction is the only way of creating money or Bitcoins in the system.

Every block must have at least one transaction. The first transaction will be a Coinbase transaction. It can be called a miner's transaction as it has a previous transaction hash of all 0's in its input. All other input type uses or refers to Bitcoins already mined or created earlier. Thus, implying that all other inputs in the current or forthcoming transactions must refer to a transaction hash of its benefactor.

To sum up, there are only two ways to own a Bitcoin; firstly, as a miner and secondly, when a Bitcoin owner sends you some. Finally, every Bitcoin will originate from a Coinbase transaction. The final parent of every Bitcoin ends with a miner. They are the federal banks.

Furthermore, if a transaction can have multiple inputs, it can also have multiple outputs. Thus, knowing the origin of a transaction hash is not enough, it is pertinent to know the index number of the output in the transaction hash of the Bitcoin supplier. The multiple outputs are generally Bitcoin addresses, like email addresses and they will receive the transacted Bitcoins. Therefore, the field following hash is the index number field, starting from 0.

The index number uniquely identifies a single output amongst numerous outputs, of the Bitcoin supplier transaction. This number is stored as a 4-byte integer.

The next couple of bytes in the Bitcoin transaction are the heart and soul of the transaction since it deliberates on ownership issues. Who can rightfully claim ownership of a Bitcoin?

First is a field that stores the length of the data that follows. In the Bitcoin literature, the ensuing data is called the Signature of the transaction. It decides on the rightful owner of the Bitcoin. In absence of this data, anyone can claim ownership of the Bitcoins.

The last field is a sequence number having a value of all F's, the largest possible value in an integer. For a long time, its value was never used, but recently along with the last field locktime, it has got a new lease of life. It takes years before some fields in the Bitcoin protocol get used on the ground.

Function `input` reads the input structure. It is given the file pointer and an input number. This function reads the data and then moves the file pointer in the file.

After reading the inputs, we read the outputs, which are relatively simple.

The variable `nooutputs` stores the number of outputs in the current transaction. This variable gets its value from the `varint` function. Accordingly, we loop through all the bytes, like the inputs. The task performing function, `doutput`, takes a file handle and an output index number and displays a single output.

The first field of the output is an important one. It is an 8-byte value in Satoshi specifying the number of Bitcoins represented by this transaction's output. Each Bitcoin can be split into 100,000,000 units. Each unit of Bitcoin, or 0.00000001 bitcoin, is called a Satoshi. A Satoshi is the smallest unit of Bitcoin.

The stored Bitcoin value is actually a very large number. The first transactions value is 5,000,000,000 i.e. 5 with nine 0's. We divide it with 1,00,000,000 or 1 with eight 0's, as the original value stored is a very small Bitcoin unit, i.e. Satoshi. The result is 50 Bitcoins.

Again, there are a set of bytes for the public key script which indirectly specify the new owner of these 50 bitcoins. The Bitcoin address is determined from this public key script.

The variable length field indicates the number of bytes to be read. We will look at the bytes representing the bitcoin address in greater detail a little later. For now, the output has a key or rules that are to be applied on the input data with the sole purpose of unlocking the Bitcoins. The true owner of the bitcoin can then be identified.

Finally, we have a concept of locking a transaction using the field locktime. The transaction usage can be determined by time, locktime and it normally has a value of 0. However, times have changed for locktime and its beginning to see some use lately.

A little later we will understand the collaborated workings of lock and key.

To sum up, there are multiple transactions stacked one above the other in a block. The fields of a transaction are the version number, the number of inputs followed by the inputs, the number of outputs followed by the outputs themselves and finally the lock time.

Each input has 5 fields, the transaction it refers to as in a hash value, the output index, script length, script content and sequence number. Sometimes the script content and script length are taken as one field.

The outputs have an easier structure, just the Bitcoin value, the script length and its contents. There are a total of 3 fields.

Displaying the Transactions of an Entire Block

Let's now display all the blocks of the downloaded first Bitcoin data file, blk00000.dat. We are getting ambitious now

For this purpose, we first copy the file from the default Bitcoin folder to the current folder.

On the Mac, we run the following command, but on windows and linux, the default folders and command will change.

```
cp ~/Library/Application\ Support/Bitcoin/blocks/blk00000.dat.
```

Now, we incorporate some small changes to the above program.

```
ch0202a.py
f = open("blk00000.dat" , "rb")
blkno = 0
while (True):
    print "-----Block Number is %d" % blkno
    magic = rint(f)
    if magic != 0xd9b4bef9:
        print ".dat file read"
        break
    print "Magic Number is %x" % magic
    size = rint(f)
    header = f.read(80)
```

Towards the end...

```
locktime = rint(f)
```

```
print "\tLockTime on transaction is %x" % locktime
blkno = blkno + 1
```

Output

```
-----Block Number is 119978
.dat file
```

The file blk00000.dat is around 128MB large and it contains a lot of the initial Bitcoin transactions.

So, we open this file and then use the while true statement in place of the for loop for an indefinite loop. The variable blkno, initially set to 0 will increment by 1 at the end of the loop. This variable also represents a record number. An error check is built in to quit out of the indefinite while loop as at some point in time we will run out of transactions to process.

In the while loop, the magic number is read as an integer and not a string and its value is checked with the predefined magic number, d9b4bef9.

If they do not match the outer most while loop is terminated. A situation like this will arise largely when all the transaction data in the block are read or when file pointer is not at the start of a new block's magic number. The blkno variable is incremented by 1 to maintain a running count of the blocks in a file.

The program quits when there is a struct unpack error.

This way we read all the 119977 blocks in the original blk00000.dat file. Further, we can display and process every Bitcoin transaction ever mined. Please press Ctrl-C or whatever keys to stop the program at any time.

The next code snippet is modified slightly.

ch0202b.py

```
print "-----Block Number is %d. File Pointer is %d" % (blkno,f.tell())

print "Magic Number is %x" % magic
size = rint(f)
print "Block Size is %d" % size
```

The size of the block is displayed, which is the size on disk the one occupied by the transaction data.

The second minor change is the use of ftell function. This function reveals the position of the file pointer in the file. In the program, it will display the start of every block in the file, blk00000.dat.

The initial Bitcoin transactions are generally Coinbase transactions, which are made for the miners and by the miners, hence they are of very little use to us. We need a Bitcoin block that has more than one transaction.

```
notran = rvarint(f)
print "Number of Transactions are %d" % notran
if notran == 2:
    break
for tno in range ( 0 , notran):
    print "Transaction number %d in block %d" % (tno , blkno)
```

The if statement simply checks the value in notran for the number of transactions. If the value is 2, we quit out of the loop and the program.

Output

```
-----Block Number is 170. File Pointer is 38032
Magic Number is d9b4bef9
```

Block Size is 490

Number of Transactions are 2

Block no 170 with a block size of 490 has two transactions

Dealing with a smaller number of transactions in a block helps in understanding Bitcoin technology better. Again, handling large files while learning becomes a bit tedious, so we extract the block having 2 transactions and save it into another file.

The following dd command performs this task.

```
>dd if=blk00000.dat of=vijay2.dat bs=1 skip=38032 count=498
```

The input file remains as blk00000.dat, our new output file is vijay2.dat. The block size is 1 to make our calculation easier.

The count variable is now the size of the block, instead of 1. The value assigned is 8 bytes larger than the block size value, 490. These 8 bytes are the bytes taken up by the magic number (4) and the size of the block (4), the initial two fields of a block.

The 170th block starts at position 38032 bytes from the start of the file and the skip parameter skips these bytes.

In the end, the dd command creates a file, vijay2.dat with count * bs bytes. The new file has a block with only two transactions, each having multiple inputs and outputs.

```
>dd if=blk00000.dat of=vijay3.dat bs=1 count=38530
```

The dd command now creates a file named vijay3.dat with the data of the first 170 blocks. The number 38530 is arrived at by adding the transaction size to the current file pointer. Block number 170 starts at position 38032 bytes, the data size of the block is 490 + 8 bytes for the initial header. This adds up to 38530.

We will use the file vijay3.dat later as we hope to understand the rest of the Bitcoin technology using only these initial 170 blocks.

ch0202c.py

```
from cfuncs import *
def doutput(f , i):
    print "\tOutput Number %d" % i
    bitcoinvalue = r8bytes(f)
    print "\t\tValue of the Bitcoin %d:%.02f" % (bitcoinvalue , bitcoinvalue * 1.0/
    100000000.00)
    outputscripplen = rvarint(f)
    print "\t\tLength of Output Script %d" % outputscripplen
    scriptpubkey = rbytes(f , outputscripplen).encode('hex')
    print "\t\tOutput Public Key Script %s" % scriptpubkey
def dinput(f , i):
    print "\tInput Number %d" % i
    prevtrhash = rhash(f)
    print "\t\tPrevious Transaction Hash is %s" % prevtrhash.encode('hex')
    print "\t\tReverse Transaction Hash is %s" % prevtrhash[::-1].encode('hex')
    outputindex = rint(f)
    print "\t\tOutput Index in transaction %x" % outputindex
```



```

inputscriptlen = rvarint(f)
print "\t\tInput Script Length ScriptSignature %d" % inputscriptlen
sigpubkey = rbytes(f, inputscriptlen).encode('hex')
print "\t\tSignature + Public Key %s" % sigpubkey
seqno = rint(f)
print "\t\tSequence Number %x" % seqno

try:
    f = open("vijay2.dat", "rb")
    blkno = 0
    while (True):
        magic = rint(f)
        print "-----Block Number is %d" % blkno
        if magic != 0xd9b4bef9:
            print ".dat file read"
            break
        print "Magic Number is %x" % magic
        size = rint(f)
        header = f.read(80)
        notran = rvarint(f)
        print "Number of Transactions are %d" % notran
        for tno in range ( 0 , notran):
            print "Transaction number %d in block %d" % (tno , blkno)
            tver = rint(f)
            print "\tVersion Number of Transaction is %d" % tver
            noinputs = rvarint(f)
            print "\tNumber of inputs are %d" % noinputs
            for i in range ( 0 , noinputs):
                dinput(f, i)
            nooutputs = rvarint(f)
            print "\tNumber of outputs are %d" % nooutputs
            for i in range ( 0 , nooutputs):
                doutput(f, i)
            locktime = rint(f)
            print "\tLockTime on transaction is %x" % locktime
        blkno = blkno + 1
except:
    print "All data read"

```

Output

```

-----Block Number is 0
Magic Number is d9b4bef9
Time is Mon Jan 12 09:00:25 2009
Number of Transactions are 2
Transaction number 0 in block 0
    Version Number of Transaction is 1
    Number of inputs are 1
    Input Number 0

```

Previous Transaction Hash is
00
Reverse Transaction Hash is
00
Output Index in transaction ffffffff
Input Script Length ScriptSignature 7
Signature + Public Key 04ffff001d0102
Sequence Number ffffffff
Number of outputs are 1
Output Number 0
Value of the Bitcoin 5000000000:50.00
Length of Output Script 67
Output Public Key Script
4104d46c4968bde02899d2aa0963367c7a6ce34eec332b32e42e5f3407e052d64ac
625da6f0718e7b302140434bd725706957c092db53805b821a85b23a7ac61725bac
LockTime on transaction is 0
Transaction number 1 in block 0
Version Number of Transaction is 1
Number of inputs are 1
Input Number 0
Previous Transaction Hash is
c997a5e56e104102fa209c6a852dd90660a20b2d9c352423edce25857fcd3704
Reverse Transaction Hash is
0437cd7f8525ceed2324359c2d0ba26006d92d856a9c20fa0241106ee5a597c9
Output Index in transaction 0
Input Script Length ScriptSignature 72
Signature + Public Key
47304402204e45e16932b8af514961a1d3a1a25fdf3f4f7732e9d624c6c61548ab5fb8cd
410220181522ec8eca07de4860a4acdd12909d831cc56cbbac4622082221a8768d1d0901
Sequence Number ffffffff
Number of outputs are 2
Output Number 0
Value of the Bitcoin 1000000000:10.00
Length of Output Script 67
Output Public Key Script
4104ae1a62fe09c5f51b13905f07f06b99a2f7159b2225f374cd378d71302fa2841
4e7aab37397f554a7df5f142c21c1b7303b8a0626f1baded5c72a704f7e6cd84cac
Output Number 1
Value of the Bitcoin 4000000000:40.00
Length of Output Script 67
Output Public Key Script
410411db93e1dcd8b8a016b49840f8c53bc1eb68a382e97b1482ecad7b148a6909a5
cb2e0eaddfb84ccf9744464f82e160bfa9b8b64f9d4c03f999b8643f656b412a3ac
LockTime on transaction is 0
All data read

The first change in this program is the introduction of try and exit clause. All code is encapsulated with it. The program will now exit gracefully. Any read beyond the end of the file throws an exception. The except clause will catch the exception and the program will quit.

The initial bytes are first checked for the magic number. Since there is a match, the code is not called and we proceed gracefully and embark on our journey.

The objective here is to understand these 2 transactions in block number 170. The file vijay2.dat has this one block and it is opened for reading. It must be noted that every transaction in a block is independent of the other.

The first transaction in the block is a Coinbase transaction that pays a miner for his efforts. It has a transaction hash of 0's and therefore, the other fields too lose their special meaning.

Let's focus on the second transaction only. There are 2 outputs in this transaction. The second output has a value of 40 Bitcoins thus indicating that someone has received these Bitcoins. The first output has only 10 Bitcoins.

The input discloses the fact that transaction hash starting with c997 and output index 0 supplied the Bitcoins for this transfer of ownership. We understand now, that this transaction hash must exist in one of the earlier blocks. This means that some block from 1 to 169 has a transaction hash starting from c997.

To confirm our hypothesis, we move to a website <https://blockchain.info/>. This is the most popular blockchain explorer on the web. This website allows us to view all the Bitcoin transactions in existence. For some reason, most websites use the big-endian format for a hash.

Our program performs this task as well. It converts the hash into a big-endian hash value. Therefore, we are displaying the same transaction hash twice. In one case, it starts with c997 and in the other case, it starts with 0437.

In the second case, we take the original value of the hash and reverse the bytes using the slice operator `[::-1]`. This original hash is not the encoded value. The slice operator takes two hex digits at a time and then reverses them.

The reverse hash value is then pasted into one of the text boxes/search box on the website. The data associated with that transaction is shown on the screen. The screen shows an error where the first transaction hash is entered in the textbox.

The website displays the transaction hash as part of block 9. This block has only one transaction, a Coinbase transaction. So, we safely conclude that 50 Bitcoins were available for a transaction in block 9, which is the miner fee.

Block 170 transfers 40 Bitcoins to one address and 10 Bitcoins to another address. At some point in time, these two outputs will be used as inputs in future. Thus, if you have no money through an output, you cannot spend that money in an input. Also, the transaction hash of this transaction has an output index of 0 as there is only output in block number 9.

Sites like blockchain.info help us to cross check our program output. They reassure our theories. We can thus verify and confirm that every input will link to a transaction hash in an earlier block.

CHAPTER 03

Computing the Merkle Hash

This chapter explains the Merkle hash in greater detail. This hash value is present in the block header and in some way, represents the hash value of the transaction data stored in the block.

Calculating the Merkle Hash for Two Transactions

```
ch0301.py
from cfuncs import *
def doutput(f , i):
    bitcoinvalue = r8bytes(f)
    outputscripflen = rvarint(f)
    scriptpubkey = rbytes(f , outputscripflen).encode('hex')

def dinput(f , i):
    prevtranhsh = rhash(f)
    outputindex = rint(f)
    inputscripflen = rvarint(f)
    sigpubkey = rbytes(f , inputscripflen).encode('hex')
    seqno = rint(f)

def hash2( a , b):
    c = a + b
    hashtemp = hashlib.sha256(c).digest()
    hashfinal = hashlib.sha256(hashtemp).digest()
    return hashfinal

try:
    f = open("blk00000.dat" , "rb")
    blkno = 0
    while (True):
        magic = rint(f)
        print "-----Block Number is %d" % blkno
        if magic != 0xd9b4bef9:
            print ".dat file read"
            break
        size = rint(f)
        header = f.read(80)
        (ver,prevhash , merklehash , time1, diff , nonce) =
        struct.unpack("I32s32sIII" , header[0:80])
```

```

    notran = rvarint(f)
    tranarray = []
    for tno in range ( 0 , notran):
        startfp = f.tell()
        tver = rint(f)
        noinputs = rvarint(f)
        for i in range ( 0 , noinputs):
            dinput(f , i)
        nooutputs = rvarint(f)
        for i in range ( 0 , nooutputs):
            doutput(f , i)
        locktime = rint(f)
        endfp = f.tell()
        f.seek( - (endfp - startfp) , 1)
        tranhash = f.read(endfp-startfp)
        hashf = chash(tranhash)
        print "Current Transaction Hash %s" % hashf.encode('hex')
        tranarray.append(hashf)
        print "Merkle Hash Block Header %s" % merklehash.encode('hex')
        if notran == 1:
            if merklehash != hashf :
                print "Transaction Count == 1. We have a mismatch at record number %d" % (blkno)
                exit()
            else:
                print "We have a Match to a T for Transaction count of %d" % notran
        elif notran == 2:
            hashf = hash2(tranarray[0] , tranarray[1])
            print "Merkle Hash calculated %s" % hashf.encode('hex')
            if merklehash != hashf:
                print "Transaction Count == 2. We have a mismatch at record number %d" % (blkno)
                exit()
            else:
                print "The Merkle Hashes Match to a T for Transaction Count %d" % notran
            else:
                print "Transaction Count is %d" % notran
            exit()
            blkno = blkno + 1
except:
    print "All data read"

```

Output

```

-----Block Number is 0
Current Transaction Hash
3ba3edfd7a7b12b27ac72c3e67768f617fc81bc3888a51323a9fb8aa4b1e5e4a
Merkle Hash Block Header
3ba3edfd7a7b12b27ac72c3e67768f617fc81bc3888a51323a9fb8aa4b1e5e4a

```

```
We have a Match to a T for Transaction count of 1
-----Block Number is 170
Current Transaction Hash
82501c1178fa0b222c1f3d474ec726b832013f0a532b44bb620cce8624a5feb1
Current Transaction Hash
169e1e83e930853391bc6f35f605c6754cfead57cf8387639d3b4096c54f18f4
Merkle Hash Block Header
ff104ccb05421ab93e63f8c3ce5c2c2e9dbb37de2764b3a3175c8166562cac7d
Merkle Hash calculated
ff104ccb05421ab93e63f8c3ce5c2c2e9dbb37de2764b3a3175c8166562cac7d
The Merkle Hashes Match to a T for Transaction Count 2
```

A transaction in Bitcoin does not and cannot have a fixed size. Nowhere in the bitcoin protocol is a field that stores the size of a transaction. So, we will have no idea on the number of transactions in a block and what's more, each transaction has a varying size.

Further, the number of transactions in a block are not part of the transactions data structures. These transactions are stored back to back in a sequential manner. Also, there is no magic number to identify the start of a transaction.

Nevertheless, in our program code, we know the start of a transaction and its end. The file position of the first field, version number of the transaction record marks the starting point and the file position after the locktime indicates the end. These positions are stored in variables `startfp` and `endfp` respectively. By storing the starting position of each transaction and it's ending position, we can easily determine the size of data in every transaction.

The variable `startfp` is assigned the position just immediately at the start of the for loop as it marks the beginning of the transaction. Similarly, the file pointer position after reading the field `locktime` is stored in variable called `endfp`. Obviously, the value in `endfp` variable will always be larger than `startfp`. The transaction data size is the difference between `endfp` and `startfp`.

With every iteration, the file pointer moves ahead by the varying size of the transaction data. The `seek` function takes us back to the start of every transaction.

The print statements in the functions used for displaying the inputs and outputs are removed as we see no reason to display the members anymore. But it is pertinent to read all the input and output fields like the script public key even when they are not used anywhere in the program. This is because the transaction size is not known in advance. The inputs and outputs are read separately.

Then, using the recently calculated size, we read the transaction data bytes into an array variable, `tranhsh`. Thereafter, the double SHA-256 transaction hash is computed using the `chash` function.

To analyze the Merkle hash, we first determine the hash of each transaction individually. They are then stored in an array. An array in Python can store anything including hashes. Since we are clueless on the number of transactions in a block, we create an empty array or list `tranarray` to store these individual hashes. It's important to have an empty array with a new block, therefore, it is initialized each time. After computing the transaction hash, we use the `append` function to add the hash into the array `tranarray`. The length or size of the array `tranarray` will reveal the count of all transactions in the block.

This program then proves that the calculated the Merkle hash matches the value in the hash field in the header.

In good old days, when we learnt programming, an error was an error. Now errors are renamed to exceptions. All code that can throw an exception is placed within `try` and `catch`. It is in bad taste for programs to quit out ungracefully, but presently we are learning and not writing code to win any contest. In our program, whenever an exception is thrown, the

code in the catch clause is executed. Here, we display an error message and then quit out. The Python programming language normally gives generic reasons for the error.

An array is used when working with a variable number of values. Using the append function, values are added to the array. One more reason to reset the array at the start of every block. In our program, the array `tranarray` stores all the computed transaction hashes. It must be noted that the sha-256 hash is stored in its un-displayable original byte form and not in its encoded version. The loop results in having all the transactions in a single block.

Now, after the loop, there are a series of if statements.

If the variable `notran` representing the number of transactions in a block, has a value of 1, then the array `tranarray` has only one transaction hash.

The computed hash in variable `hashf` will have the same value as in the variable `merklehash`, the transaction hash in the block header. In case of only one transaction in a block, the variable `hashf` and the array member `tranarray[0]` will have the same value. If there is a mismatch, we display an error.

It is very simple to calculate Merkle hash for one transaction in a block, it is equal to the hash of the transaction itself. The problem arises when there are two or more transactions in a block.

The Merkle hash is nothing but a binary tree of hashes. The hash of multiple hashes is calculated following certain rules, and thus it becomes more complex.

Let's take the case of a block having two transactions only, block 170 in the .dat file. First, to make our jobs easier, we create a function called `hash2`.

The array element `tranarray[0]` and `tranarray[1]` have the hashes of first and second transactions in the block. The Merkle hash algorithm requires two hashes of two transactions. These two hashes are then concatenated to form a string. Finally, a double hash, as in SHA-256 of the resultant string is taken.

We repeat, in Merkle hash, two hash strings are concatenated and a new hash value of the resultant string is calculated. All our hashes are bytes from 0 to 255.

For reasons best known to the originators, the Bitcoin ecosystem does not take a hash of the entire transaction data. It first calculates the individual hashes of each transaction and then concatenates these hashes, but only two at a time.

So, if the number of transactions are 4 then we first compute two hashes, first one comprising transaction 1 and 2, let's say A. This is followed by the second hash of transactions 3 and 4, let's say B. The Merkle hash is the hash of these two subsequent hashes, A and B.

A small milestone reached as we can calculate the Merkle hash for 2 transactions in a block.

There are few modifications when there are more than 2 transactions in a block. The odd and even number of transactions are also handled in a different manner. So, let's look at these challenges in the next program.

Finding the Merkle Hash of a Block with 6 Transactions

```
ch0302.py
from cfuncs import *
def doutput(f , i):
    bitcoinvalue = r8bytes(f)
    outputsriptlen = rvarint(f)
    scriptpubkey = rbytes(f , outputsriptlen).encode('hex')
def dinput(f , i):
    prevtranhsh = rhash(f)
```

```
    outputindex = rint(f)
    inputscripflen = rvarint(f)
    sigpubkey = rbytes(f, inputscripflen).encode('hex')
    seqno = rint(f)

def hash2( a , b):
    c = a + b
    hashtemp = hashlib.sha256(c).digest()
    hashfinal = hashlib.sha256(hashtemp).digest()
    return hashfinal

#try:
f = open("blk000000.dat" , "rb")
blkno = 0
while (True):
    magic = rint(f)
    print "-----Block Number is %d" % blkno
    if magic != 0xd9b4bef9:
        print ".dat file read"
        break
    size = rint(f)
    header = f.read(80)
    (ver,prevhash , merklehash , time1, diff , nonce) =
    struct.unpack("I32s32sIII" , header[0:80])
    notran = rvarint(f)
    tranarray = []
    for tno in range ( 0 , notran):
        startfp = f.tell()
        tver = rint(f)
        noinputs = rvarint(f)
        for i in range ( 0 , noinputs):
            dinput(f , i)
        nooutputs = rvarint(f)
        for i in range ( 0 , nooutputs):
            doutput(f , i)
        locktime = rint(f)
        endfp = f.tell()
        f.seek( - (endfp - startfp) , 1)
        tranhash = f.read(endfp-startfp)
        hashf = chash(tranhash)
        print "Current Transaction Hash %s" % hashf.encode('hex')
        tranarray.append(hashf)
        print "Merkle Hash Block Header %s" % merklehash.encode('hex')
        if notran == 1:
            if merklehash != hashf :
                print "Transaction Count == 1. We have a mismatch at record number %d" % (blkno)
                exit()
            else:
```



```

print "We have a Match to a T for Transaction count of %d" % notran
elif notran == 2:
    hashf = hash2(tranarray[0] , tranarray[1])
    print "Merkle Hash calculated %s" % hashf.encode('hex')
    if merklehash != hashf:
        print "Transaction Count == 2. We have a mismatch at record number %d" % (blkno)
        exit()
    else:
        print "The Merkle Hashes Match to a T for Transaction Count %d" % notran
elif notran == 3:
    round1 = hash2(tranarray[0] , tranarray[1])
    round2 = hash2(tranarray[2] , tranarray[2])
    roundf = hash2(round1 , round2)
    print "merkle Root is %s" % merklehash
    if merklehash != roundf:
        print "Transaction Count == 3. We have a mismatch at record number %d" % (blkno)
        exit()
    else:
        print "The Merkle hash matches to a T for Transaction Count %d" % notran
elif notran == 4:
    round1 = hash2(tranarray[0] , tranarray[1])
    round2 = hash2(tranarray[2] , tranarray[3])
    roundf = hash2(round1 , round2)
    print "Merkle hash is %s" % roundf.encode('hex')
    if merklehash != roundf:
        print "Transaction Count == 4. We have a mismatch at record number %d" % (i - 1)
        exit()
    else:
        print "The Merkle hash matches to a T for Transaction Count %d" % notran
elif notran == 5:
    round1 = hash2(tranarray[0] , tranarray[1])
    round2 = hash2(tranarray[2] , tranarray[3])
    round3 = hash2(tranarray[4] , tranarray[4])
    round4 = round3
    round5 = hash2(round1 , round2)
    round6 = hash2(round3, round4)
    roundf = hash2(round5 , round6)
    if merklehash != roundf:
        print "Transaction Count == 5. We have a mismatch at record number %d" % (i - 1)
        exit()
    else:
        print "The Merkle hash matches to a T for Transaction Count %d" % notran
elif notran == 6:
    index = 0
    cnt = notran / 2
    round = []
    for i in range ( 0 , cnt ):

```

```
round.append(hash2(tranarray[index] , tranarray[index+1]))
index = index + 2
round.append(round[cnt-1])
print cnt , len(round)
roundarr = []
index = 0
for i in range(0 , len(round) / 2 ):
roundarr.append(hash2(round[index] , round[index+1]))
index = index + 2
roundfinal = hash2(roundarr[0], roundarr[1])
print "Merkle hash is %s" % roundfinal.encode('hex')
if merklehash != roundfinal:
print "Transaction Count == 6. We have a mismatch at record number %d" % (i - 1)
exit()
else:
print "The Merkle hash matches to a T for Transaction Count %d" % notran
else:
print "Transaction Count is %d. Quitting" % notran
exit()

blkno = blkno + 1
#except:
# print "All data read"
```

Too much Output, None displayed

Let's take a small step forward where there are 3 transactions in a block. As per the Merkle rulebook, first the hash of transactions 1 and 2 is calculated. Let's say round1 is the hash of transaction 1 and transaction 2.

Now for transaction 3, a new hash is created by adding the hash of transaction 3 to itself. The rulebook could have allowed the use of the leftover hash by itself but instead it suggests creating a new hash by adding the hash to itself. Thus, round 2 is the hash of transaction 3 concatenated with transaction 3.

Finally, the hash of concatenated round1 and round2 is calculated. This is the resultant Merkle hash value and in our code, it is stored in variable roundf. For an odd number of transactions, we add the same hash to itself and create an even number of transactions.

For 4 transactions, it is much simpler. The first two transactions give one hash and the next two give the second hash. The hash is now calculated on the two new values.

Now if the number of transactions is 5: First, the hash of transaction 1 and transaction 2 is determined (in variable round1), then the hash value of transaction 3 and transaction 4 is computed (variable round2), and finally it is the hash value of transaction 5 and again transaction 5(variable round3).

The result however, gives an odd number of hashes. So, we create one more hash round4 which is the hash of round3 with itself. There are an even number of hashes as we are now left with 4 hashes. It is eventually reduced to 2 hashes and then to a single hash value, the Merkle hash.

If there are 6 hash transactions in the array, then initially 3 rounds of hashes are computed, round 1 is transaction 1,2 then round 2 with transactions 3,4 and round 3 with transactions 5,6. These hashes are further saved in an array called round. Since there are 3 hashes, one more copy of the last hash is made to arrive at an even number of 4 hashes. These

four hashes are further hashed, two at a time, to give the last two hashes. The latest 2 hashes are stored in another array called roundarr. Finally, the last two hashes give the merkle hash value of all the 6 transactions in the block.

If there are 12 hashes in the transaction array, we hash them two at a time to get 6 hashes. Then the same concept is repeated as explained above to get a single hash value.

We could go on ad infinitum but the concept remains the same. Keep reducing the number of hashes by half, but if we arrive at an odd number of hashes, then hash the hash with itself. This gives an even number of hashes and ultimately the final hash value.

In our program, we tried to automate some of these hash calculations using two sets of loops. Since the code is repetitive, it can be easily placed in a loop, though it may not be the best fit. But, in the long run, we need a more generic approach to handle this recursion.

Open the file vijay2.dat in place of blk00000.dat and the code will stop running after successfully calculating over 38000 blocks.

The above program cannot take care of all the possible number of transactions in a block. Obviously, there should be a way to end the infinite calling process depending upon some condition being met, otherwise the program runs forever.

The main objective here was to introduce and understand recursion better. In recursion, the same function is called repeatedly with different parameter values.

Let's use this knowledge to write a much smaller program in an elegant programming style using recursion.

Calculating the Merkle hash using recursion

```
ch0303.py
from cfuncs import *
def doutput(f , i):
    bitcoinvalue = r8bytes(f)
    outputscripflen = rvarint(f)
    scriptpubkey = rbytes(f , outputscripflen).encode('hex')

def dinput(f , i):
    prevtranhsh = rhash(f)
    outputindex = rint(f)
    inputscripflen = rvarint(f)
    sigpubkey = rbytes(f , inputscripflen).encode('hex')
    seqno = rint(f)

def hash2( a , b):
    c = a + b
    hashtemp = hashlib.sha256(c).digest()
    hashfinal = hashlib.sha256(hashtemp).digest()
    return hashfinal

def rmerkle(arr):
    if len(arr) == 1:
        return arr[0]
    emptyarray = []
    for i in range(0, len(arr)-1, 2):
        newhash = hash2(arr[i], arr[i+1])
```

```
    emptyarray.append(newhash)
    if len(arr) % 2 == 1:
        newhash1 = hash2(arr[-1], arr[-1])
        emptyarray.append(newhash1)
    return rmerkle(emptyarray)

try:
    f = open("blk00000.dat", "rb")
    blkno = 0
    while (True):
        magic = rint(f)
        if magic != 0xd9b4bef9:
            print ".dat file read"
            break
        size = rint(f)
        header = f.read(80)
        (ver,prevhash , merklehash , time1, diff , nonce) =
        struct.unpack("l32s32slll", header[0:80])
        notran = rvarint(f)
        tranarray = []
        for tno in range ( 0 , notran):
            startfp = f.tell()
            tver = rint(f)
            noinputs = rvarint(f)
            for i in range ( 0 , noinputs):
                dinput(f , i)
            nooutputs = rvarint(f)
            for i in range ( 0 , nooutputs):
                doutput(f , i)
            locktime = rint(f)
            endfp = f.tell()
            f.seek( - (endfp - startfp) , 1)
            tranhash = f.read(endfp-startfp)
            hashf = chash(tranhash)
            tranarray.append(hashf)
            print "Merkle Hash Block Header %s" % merklehash.encode('hex')
            merklehashn = rmerkle(tranarray)
            print "New Merkle Hash is %s" % merklehashn.encode('hex')
            if merklehashn == merklehash :
                print "We have a match at Block Number %d" % blkno
            else:
                print "We have a mismatch at record number %d Transaction Count is %d" % (blkno, notran)
            exit()
            blkno = blkno + 1
except:
    print "All data read"
```

Output

Merkle Hash Block Header

ff104ccb05421ab93e63f8c3ce5c2c2e9dbb37de2764b3a3175c8166562cac7d

New Merkle Hash is

ff104ccb05421ab93e63f8c3ce5c2c2e9dbb37de2764b3a3175c8166562cac7d

We have a match at Block Number 170

Merkle Hash Block Header

3567bd81754bc529248537c69ae13a37fe191e6888545bb33e60effcd8853146

New Merkle Hash is

3567bd81754bc529248537c69ae13a37fe191e6888545bb33e60effcd8853146

We have a match at Block Number 119977

.dat file read

As we mentioned earlier and all the time, the number of transactions held in a block is never known nor is the size of every transaction. What we do know is that to calculate the merkle hash of these transaction there is a lot of repetition. Earlier, we used a for statement to repeat code but now we want to repeat a function itself, so we use recursion.

Our main objective is to arrive at the merkle hash value and for this purpose, a new hash value is created repeatedly by joining two hashes. Also, an odd number of hashes is treated very differently from an even number of hashes.

Let's assume there are 100 transactions in a block. We use the old approach of looping. Joining the two adjacent hashes using function hash2 will give 50 hashes in hand. On repeating the same procedure, these 50 hashes become 25 hashes. Now comes the problem, we cannot repeat the above process as we have an odd number of hashes. Therefore, at every reiteration, a check is performed to verify the tally to be an odd number of hashes or even. Accordingly, an extra hash which is simply a copy of the last hash is added to the list of hashes. We now get 26 hash's, an even number of hashes. As we now have an even number, the above process is repeated. But now again, there are 13 hashes. An odd number again. We once again add one more hash at the end and repeat the process until we have one hash left.

The function rmerkle uses recursion to perform the same task. This function is passed an array, tranarray, containing all transaction hashes.

A check is performed at the very beginning of the function. If the length of the tranarray (now called arr) is 1 then the function returns the hash value in the first or the 0th element of the array. An array index in python starts from 0 and not 1. This is the Merkle hash value of the transactions in the block. Also, this situation will arise when there is only one transaction in the block, and when all the hashes have been concatenated together. A clean exit. Every recursive function is like an indefinite loop, it must have an exit point.

In cases where the length is not 1, we create an empty array aptly called emptyarray. As this is a local variable, it is reinitialized every time the function gets called. However, each function call retains its own set of local variables during its execution. There will be multiple copies executing in the process but they all will have a different state.

The only for loop in the function now gets called with a step of 2, which is the third parameter. When we have 100 transactions in the array tranarray, the array index is from 0 to 99. Along the same lines, the range function does not take the last number into account. The step parameter in the for loop increments the variable by 2 each time. So, the value of i now goes from 0 to 98 in steps of 2, i.e. 0, 2, 4 etc.

First time around, the variable i has a value of 0. So, the hash of first two adjacent hashes as in hash arr [0] and arr [1] is computed. Then it will be arr[2] and arr[3] and finally in the end it will be arr [98] and arr [99]. The variable i has a value of 98 in the last round.

The newly minted sha-256 hash is saved in a variable newhash and then added to the local array, emptyarray. This task is performed 50 times. Thus, in one swoop, the first function call to the rmerkle function will store 50 members in the array, emptyarray.

Once done, the same recursive function rmerkle is called the second time but with the newly minted array emptyarray. This array has 50 hashes. When the for loop ends, there will be 25 hashes in emptyarray, which is unfortunately an odd number.

The % operator is the mod operator, it gives a remainder. When a number % 2 is 1, the number is an odd number. Similarly, number % 2 with a remainder of 0 is an even number. The if statement is true when there are an odd number of hashes in the array.

Here, we create a hash by using the last hash twice. The last hash is deciphered using [-1] or the len function. The variable newhash1 is given the resultant hash which is then added to the array, emptyarray.

At some point in time, the function rmerkle will be called with 2 hashes in the array. This will result in an array of size 1. When the function rmerkle is called again, the first if statement is true and the Merkle hash of the only member in the array is returned.

This last Merkle hash is saved in a variable merklehashn and then compared with the one in the Block header, merklehash. If they do not match, we exit out after displaying an error.

This is much cleaner code explaining the use of recursion to calculate the hash value.

An Alternative Method using Recursion

ch0303a.py

```
def rmerkle(arr):
    if len(arr) == 1:
        return arr[0]
    emptyarray = []
    if len(arr) % 2 == 1:
        arr.append(arr[-1])
    for i in range(0, len(arr)-1, 2):
        newhash = hash2(arr[i], arr[i+1])
        emptyarray.append(newhash)
    return rmerkle(emptyarray)
```

There is no one way of writing code. In this code snippet, an odd size array is made even by appending one member. Then the for statement is executed in a loop. Our view is recursion always wins.

This chapter has given us an in-depth understanding of the Merkle hash and its computation techniques.

Though the question that bothers us is why use a Merkle hash? Since the block size cannot be larger than 1MB, why not take the hash of this data in one go. Some mysteries may never be solved.

In the world of Ethereum, they have extended the Merkle tree further.

CHAPTER 04

Bitcoin Addresses

Bitcoins are transferred from one Bitcoin address to another. Before we understand the outputs in a transaction, let's spend some time on Bitcoin addresses.

To call a person, we must own a phone/mobile number and have the other person's number. Similarly, to send an email, we need an email id and we require the destined fellow's email address. In the same vein, to transfer Bitcoins, we need a Bitcoin address at both ends. To a cynic, it is simply transferring ownership of money from one Bitcoin address to another. Bitcoin users are constantly looking for more information on the procedures and techniques used, to generate a valid Bitcoin address. Since creating one is a tedious task, the commonly used practice is to download some wallet software and spawn multiple Bitcoin addresses.

In this chapter, we will first use a library to understand the Bitcoin concepts and then generate Bitcoin addresses. Thereafter, we will replace the library code with our code.

Command prompt \$: `sudo pip install pybitcointools`

The above command will install a Python library `pybitcointools`, written by Vitalik Buterin. This library is used to generate Bitcoin addresses. Vitalik Buterin is the unofficial God of the crypto coin's universe. `Pybitcointools` may be the best Bitcoin Python library out there but Vitalik is best known for his work as the lead inventor on Ethereum, also known as Bitcoin 2.0.

Using functions from a Python library to Generate a Bitcoin Address

```
ch0401.py
import pybitcointools
privkey = '4200000000000000000000000000000000000000000000000000000000000000'
print "The length of the Private Key is %d. Key is %s" % (len(privkey) , privkey)
pubkey = pybitcointools.privtopub(privkey)
print "The Length of the Public Key is %d. Key is %s" % ( len(pubkey) , pubkey)
hash = pybitcointools.hash160(pubkey.decode('hex'))
print "The RipeMD hash length is %d. Hash is %s" % (len(hash) , hash)
baddr = pybitcointools.hex_to_b58check(hash)
print "After the b58check or actual Bitcoin address is %s" % baddr
print "The Length of the Bitcoin Address is %d" % len(baddr)
```

Output

```
The length of the Private Key is 64. Key is
4200000000000000000000000000000000000000000000000000000000000000
The Length of the Public Key is 130. Key is
04f9a7699db2eeca4116373fde8e5c7f46af3d81c96b912cdc0e2dd67d00ed6d
```

```
7081888bc7c39617d4f7c3437b4ace461402174ed76561090838123f952d41488
The RipeMD hash length is 40. Hash is
56acc14dedbdb502d3e1dd23ca13b4cad4d110bd
After the b58check or actual Bitcoin address is
18uJ4jT1wbVC6SahYGxkTTf35xwQ7MkV66
The Length of the Bitcoin Address is 34
```

Let's learn how to create a Bitcoin address by using some functions from a library.

A fair warning, it's a million times more difficult than creating an email address. The reason being, there are no rules as such to follow when generating an email address whereas for Bitcoin addresses, we need a full chapter. Every Bitcoin address must be unique. There is a lot of hard core mathematics and cryptography used in the background when creating one address. Somewhat similar when creating the SHA-256 hash. In the program, we simply call a function and the hash is calculated.

Let's look at the concept of a private key. It is a number not disclosed to anyone. The moniker is key and not number because generally keys are difficult to remember compared to numbers. Though a number and key mean the same.

The private key in our program is a string with a value of 420 and many 0's, the key length must be 32 bytes or 64 digits. It is a good security practice to have a very long number as a key. In India, there is a criminal section of the law that defines cheating and this section is numbered 420. Also, a crook or thief is called a 420.

The pybitcointools library has laid down stringent rules on the size/length of a private key. If it is not followed, an exception is thrown. We simply adhere to it, even though we believe these restrictions are arbitrary. When we write our own code a little later, the private key will be one or two digits only.

Generally, a function is used to generate a 32-byte random number. Then, this random number is hashed using the SHA-256 hash algorithm. The end-result is another 32-byte number which can be ultimately used as a private key. This way, no two people in the universe would get the same private key. Plus, it is more difficult to crack. A private key is not to be shared, it's to be kept private, as the name suggests.

Cryptography works on the principle of encrypting and decrypting. One key encrypts and the other key decrypts or reverses the encryption. We label these keys as private or public key. The key that is kept private is called the private key and the key that is made public is called the public key. The two keys look the same.

Now let's move ahead and generate a public key from the private key.

In conventional RSA public key cryptography, the private key and public key are chosen at the same time. In Bitcoin, the public key is derived from the private key. Given a public key hash, it is impossible to determine the private key. It's a one-way operation. But, deriving a public key from a private key is very much possible and very easy also.

Bitcoins use the mathematics behind the Elliptic Curve Cryptography (ECC) to generate the public key. The pybitcointools module has a function called `privtopub` which takes a private key as a parameter and it generates the public key.

The chapter on cryptography has more mathematics on the algorithm used for generating public key.

For now, we first generate the private key and then the Elliptic Curve mathematics takes over to generate a public key. The public key is 65 bytes long. Since we are displaying it as a hex string, the length is 130 digits.

Time and again, it must be noted that given a private key, we will be able to determine the public key but given a public key, it is impossible to arrive at the private key. The transformation is one way only.

It makes no sense at all to display the encoded public key as the actual individual bytes have no meaning.

The decode function can decode the public key, a string of hex bytes, to the original key which is 65 bytes large. But

the value of the decoded public key cannot be displayed and evaluated. Some functions take a hex string as input whereas some take the original decoded value only.

The public key could have been designated as a Bitcoin address. But the Bitcoin world decided against it and rightly so. The 65 byte long public key is given to the SHA-256 hash function, that returns a unique 32-byte number or 256-bit number. The 65-byte key is reduced to 32 bytes. No two different public keys will give the same hash value. This is called the property of collision.

Also, an extra level of security is added since using the hash value it is difficult to determine the public key. Thus, it takes away every chance or hope of obtaining the private key. In effect, there are two levels of security to the private key, the SHA-hash and the public key.

Thereafter, another hashing method called RipeMD160 is used to get a smaller hash. In the end, the hash key size is only 20 bytes large.

This hashing task is executed internally by the function `hash160` from the library.

The function from the `pybitcointools` library performs two jobs, it first computes the SHA256 hash and then computes the RIPEMD160 hash. We know this because we read the source code of all libraries.

Our pea sized brain can never understand the theory behind choosing two different types of hash methods. Normally, Bitcoins use the same sha-256 hash twice over but it is believed that using two different hash techniques make the key somewhat more secure.

This public key is our Bitcoin address. Nevertheless, we yet cannot announce to the world that our Bitcoin address is `56acc14dedbdb502d3e1dd23ca13b4cad4d110bd`. This is because while typing or advertising this number as our Bitcoin address, chances are that we will make errors. Some fonts make different characters look the same. So, a checksum is calculated and added at the end of the Bitcoin address.

The bigger problem is how do you advertise this address. With non-printable characters as part of a Bitcoin address, how do you print characters on a screen if they are not displayable. Moreover, a 10 or 13 would be interpreted as a new line.

Let's take a brief look at the workings of the Base64 and Base58 encoding.

A normal ASCII string has numbers from 0 to 255. Earlier, the Internet used an encoding called Base64 encoding wherein binary files were sent across using numbers ranging from 0 to 63. The 64 in Base64 represented the number of characters that can be used, 26 small letters, 26 capital letters and then 10 digits plus 2 more characters. Thus, the ASCII file was encoded using 64 characters and then sent across. This encoding has nothing to do with compression or encryption.

Most programs display a new line with numbers like 10 or 13. We cannot have new lines show up in the middle of a Bitcoin address. Ditto for an email address.

The Base58 encoding structure goes a step further and removes the `+` and `-` from the list of valid characters. Most importantly, it does not use the `O` (capital O) and `0` (zero) as they look alike. The other pair that looks similar and can be confusing to the eye is the `I` (capital I) and `i` (small I). The base58 encoding removes these 6 characters from the Base64 encoding. This Base58 encoding was invented by Mr. Satoshi.

Bitcoin Addresses use the Base58 encoding, thus using only 58 characters. The end-result or the size of the string is also smaller after the encoding. Plus, there is a checksum at the end of the address as an added error check. A Bitcoin address with a checksum which was originally 40-digits large, now takes up only 34 digits. We will explain later, how this checksum is tagged to the end of the Bitcoin address.

A reminder. The public key is hashed using the SHA 256 algorithm and it has a length of 32 bytes. Then, the RipeMD

hash is used which in turn gives us a 20-byte hash. This 20-byte hash is converted to a Base58 encoding string with a checksum tagged at the very end.

Thus, our Bitcoin address is a Base58 encoding of a hashed public key.

In our program, the Bitcoin address finally is 18uJ4jT1wbVC6SahYGxkTTf35xwQ7MkV66. A disclaimer here: This address is not to be used for any transactions, it is only used to explain a concept.

The website blockchain.info is very intelligent. It can distinguish between various bitcoin artifacts like Block Header hashes, transaction hashes and Bitcoin addresses as well. We simply paste our newly calculated Bitcoin address into the textbox on blockchain.info. To our surprise, there are a whole lot of transactions with this Bitcoin address, 8 transactions to be precise and the balance is 0. Maybe one or two transactions are ours but not the others.

This re-validates the fact that the above code generates a valid Bitcoin address when it is given a random private key. This is the simplest Bitcoin address one can ever generate.

The Bitcoin address starts with a 1. We will explain later how to generate a Bitcoin address starting with a 3.

Using a library is great but all code in libraries is a black box to understanding. So, let's now look at creating Bitcoin addresses using our code. We will do some strange things in a couple of programs. We will take a string, let's assume it is our Bitcoin address, do some multiplications to convert it into a real number and then we will convert this number back to the original string.

All this is Greek and Latin presently, so let's understand some more concepts with code.

Our First Stab at Encoding and Decoding Numbers

```
ch0402.py
baddr = '2'
v = '123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'

rtot = v.index(baddr[0])
print "Final Base58 Total as Long %d" % rtot

(ans , r ) = divmod(rtot , 58)
print "Initial answer %d remainder %d" % (ans , r)
f = v[r]
print "After while answer %05d remainder %03d char %s final str %s" % (ans, r, v[r] , f)
f = f[:-1]
print(f)
```

Output

```
Final Base58 Total as Long 1
Initial answer 0 remainder 1
Answer 00000 remainder 001 char 2 final str 2
2
```

In the above program, the variable baddr has a value of 2 which is presumed to be a bitcoin address. The variable v stores the 58 characters that are part of a valid Base58 set.

The ordering of the valid characters in the Base58 encoding are decided by the Bitcoin standard. First the numbers, 1 to 9 then the capital letters (less 2) and finally the lower-case letters (less 1). Reordering the numbers to the end of the string v works for our code, but it surely will not work with a real Bitcoin address.

A string in Python works like an array. Therefore, `baddr[0]` is 2. The string class also has an index function that returns an offset of the character from the string, namely `v`. Thus, we are looking at `v.index(2)`. The character 2 is at the offset of 1 in the string, the counting starts from 0, therefore the variable `rtot` has a value of 1.

The `divmod` method returns the result and the remainder after a modular division. We divide the value in variable `rtot`, 1 with the number 58. The answer, `ans` is 0 and the remainder, `r` is 1.

We now reverse the entire process. We find the character placed at position `r` in the string `v`. The value of variable `r` is 1 so the character is 2. Reversing the string has no effect in this case as there is only one character.

So, all that we did was take a string of size 1 and locate the character at its position in the base58 string of characters. We then reverse the process. Though artless, it is the heart of our code. Let's make slight modifications to the above code.

A Second Stab at Understanding Encoding and Decoding

```
ch0403.py
baddr = '2'
v = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'

rtot = 0
for c in baddr:
    rtot = rtot * 58 + v.index(c)
    print "Running Base58 Character %c Total long %d Index %d" % (c , rtot , v.index(c))
print "Final Base58 Total as Long %d" % rtot

f = ''
(ans , r ) = divmod(rtot , 58)
print "Initial answer %d remainder %d" % (ans , r)
while r:
    f = f + v[r]
    (ans, r) = divmod(ans, 58)
    print "After while answer %05d remainder %03d char %s resultstr %s" % (ans, r, v[r] , f)

f = f[::-1]
print(f)
```

Output

```
Running Base58 Character 2 Total long 1 Index 1
Final Base58 Total as Long 1
Initial answer 0 remainder 1
After while answer 00000 remainder 000 char 1 resultstr 2
2
```

As before, string variable `v` stores the 58 valid characters of a bitcoin address and the variable `baddr` contains a bitcoin address. The address will keep increasing in length as time goes by. The program convert this bitcoin address string to a very large number and thereafter back to a bitcoin string. We will decode and encode a string.

The variable `rtot` is initialized to 0 and it will contain a running total. This running total is computed from the placement or index of the bitcoin address characters in the string `v`.

A for loop iterates through every character in the string, `baddr`.

Initially, we have a single character as our Bitcoin address so the program works and behaves like the previous one. The variable `c` contains the only character, i.e. 2 in the string `baddr`. We look up the position of the character/number 2 in the string `v`. The `index` function returns 1 as it is the value in the second position in the string.

The value in the variable `rtot` is multiplied by 58. Since the `rtot` variable has a value of 0, the answer is 0. Then in the same breath, this resultant value, 0 is added to the index value, 1. The variable `rtot` now has a value of $0 + 1 = 1$.

The variable `rtot` is multiplied with 58 because it is Base58 encoding and there are 58 different characters to deal with.

As the string `baddr` is only one character, the `for` loop ends and the value in the `rtot` variable stays at 1. The `divmod` function divides 1 by 58, and returns a tuple. The variable `ans` has a value of 0 and then the remainder `r` has a value of 1.

In the last `while` loop, the remainder 1 is searched in the `v` string. It is the second character in the string and the value is 2 i.e. the character 2. The variable `f` joins all these characters.

In the `divmod` function, the value of the variable `ans`, 0 is divided by 58. The answer is 0 and the remainder, `r` is also 0. Depending upon the value in the remainder variable `r`, the loop continues or exits. The loop ends thanks to the remainder variable `r` being 0.

By first decoding and then encoding, we come back to the same Bitcoin address we started with. A number is calculated and then from that number, the original Bitcoin address is obtained.

The running total variable, `rtot` in some way, represents the Bitcoin address. Once again, the string functions, `decode` and `encode`, work in a very different way in real life. The outputs of these function are of type string. In our case, `decode` returns a number whereas `encode` takes a number as a parameter, input but returns a string.

Let's only change one line in the above code.

```
baddr = '2A'
```

Output

```
Running Base58 Character 2 Total long 1 Index 1
Running Base58 Character A Total long 67 Index 9
Final Base58 Total as Long 67
Initial answer 1 remainder 9
After while answer 00000 remainder 001 char 2 resultstr A
After while answer 00000 remainder 000 char 1 resultstr A2
2A
```

As before, the variable `rtot` will have the value of 1 at the end of the first iteration of the `for` loop. In the second iteration, the value of `rtot` which is 1 is multiplied with 58 and then 9 is added to its result. This is because character A is at the 10th position in the `v` string variable and the value in the tenth position is 9. The variable `rtot` now has a value of 67.

The bitcoin address 2A is now equal to a number whose value is 67.

The weight given to numbers is smaller than the weight given to the letters of the alphabet. The index values are smaller for a number but they are largest for the lower-case letters. Thus, the value of a two-character bitcoin address will change depending upon whether it has more lower case letters or less. Greater the characters at the end of the base58 character string `v`, greater the value of the variable `rtot` that represents the Bitcoin address.

Now let's get back to encoding the Bitcoin address. We have a number in `rtot`, a Bitcoin address and to display it, we must encode it. As a rule, always encode the data before displaying.

So, let's convert the value of 67 to a bitcoin address of 2A. We are effectively going to reverse what we did earlier. Before

entering the while statement, we divide 67, the value of variable `rtot` by 58. This gives 1 as the answer and 9 as the remainder.

The 10th character (remainder is 9) in the string variable `v` is the character A. This is the last character of the bitcoin address. The remainder variable `r` is used to represent each bitcoin character address.

In the while loop, the `divmod` function gives the remainder as 1 and the answer is 0. This remainder is used as an index in the `v` string, which is the character 2.

Joining or concatenating the two values results in a string, A2. This is the reverse of the original Base58 string, 2A. Therefore, we reverse the string using the slice operator.

Let's take one more example,

```
baddr = '2A3'
```

Output

```
Running Base58 Character 2 Total long 1 Index 1
Running Base58 Character A Total long 67 Index 9
Running Base58 Character 3 Total long 3888 Index 2
Final Base58 Total as Long 3888
Initial answer 67 remainder 2
After while answer 00001 remainder 009 char A resultstr 3
After while answer 00000 remainder 001 char 2 resultstr 3A
After while answer 00000 remainder 000 char 1 resultstr 3A2
2A3
```

We now change the bitcoin address to 2A3.

The first value of the variable `rtot` is 1 for the character 2. Then with the next character being A and the multiplication by 58, the value of `rtot` becomes 67 like before.

Now with the advent of the character 3, 67 is multiplied by 58. To the resultant value, 3886, we add the value currently positioned at the offset of 3 in string `v`. The value is 2. The variable `rtot` stores the final value of 3888. As the characters in the Bitcoin address increases, so does the final value of the `rtot` variable.

Once again, the principle is very simple. We convert the base58 string to a large number by multiplying the running total by 58. Then we added the position of the character in the base58 string `v`. Thus, we divide by 58, and we get the same result but now backwards.

First comes 3, then A and then 2. The program code thus gives an answer of 3A2 which is reversed to come back to the original.

This program can be modified very easily to encode and decode a string into any base from 0 to 255.

Encoding and Decoding using Base256 Encoding

```
ch0404.py
baddr = '2A3'
v = ''.join([chr(x) for x in range(256)])
rtot = 0
for c in baddr:
    rtot = rtot * 256 + v.index(c)
    print "Running Base256 Character %c Total long %d Index %d" % (c , rtot , v.index(c))
```

```
print "Final Base256 Total as Long %d" % rtot
f = ''
(ans , r ) = divmod(rtot , 256)
print "Initial answer %d remainder %d" % (ans , r)
while r:
    f = f + v[r]
    (ans, r) = divmod(ans, 256)
    print "After while answer %05d remainder %03d char %s resultstr %s" % (ans, r, v[r] , f)
f = f[::-1]
print(f)
```

Output

```
Running Base256 Character 2 Total long 50 Index 50
Running Base256 Character A Total long 12865 Index 65
Running Base256 Character 3 Total long 3293491 Index 51
Final Base256 Total as Long 3293491
Initial answer 12865 remainder 51
After while answer 00050 remainder 065 char A resultstr 3
After while answer 00000 remainder 050 char 2 resultstr 3A
After while answer 00000 remainder 000 char resultstr 3A2
2A3
```

The above program is nearly identical to the previous one. The only difference is that, in base256 encoding, all the valid characters are part of the character-set, from 0 to 255. It is not possible to display the entire character set because most of them are not printable.

We could have written this code over multiple lines but the Python programming language makes it simpler. The join function is part of the string class, it is used to join or concatenate strings. The chr function is opposite of ord, it returns the character that corresponds to a number. The for command can be used anywhere in the program and it looks at a range of values from 0 to 255.

The rtot value is multiplied by 256 for base256 encoding and to reverse the string, it is divided by 256. The variable rtot's values are much larger with base256 encoding compared to base58 encoding. Besides these few modifications, nothing else changes.

```
ch0405.py
s = '41'
print s.decode('hex')
s = '4142'
print s.decode('hex')
s = '123'
print s.decode('hex')
```

Output

```
A
AB
TypeError: Odd-length string
```

Why are these error messages being displayed? Let's understand the reasons.

The string `s` has a value of `41`. When this string is decoded, `41` is the hex value and its equivalent in decimal is `65`. The character displayed is `A` from the ASCII character set. Similarly, the hex value of `66` is `42` and its ASCII value is `B`.

When the length of the string is an odd number, there is confusion. The decode function can handle the `12`, but it expects two digits next. Since there is only one digit, `3` an exception is thrown. When a string is decoded, its size or length must be an even number.

It is difficult to figure out an answer when decoding a base58 encoded Bitcoin address. As we mentioned earlier, the smaller case characters generate larger numbers. If we try and convert this number into a string, we have no idea what the resultant string length may be. This could result in an odd numbered string which cannot be decoded.

Converting Strings into Numbers and Numbers to Strings

```
ch0406.py
v = '123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
def decode(baddr):
    rtot = 0
    for c in baddr:
        rtot = rtot * 58 + v.index(c)
    return rtot

def encode(rtot):
    f = ''
    (ans, r) = divmod(rtot, 58)
    while r:
        f = f + v[r]
        (ans, r) = divmod(ans, 58)
    f = f[::-1]
    return f

a = decode("ABab")
print encode(a)

print "%x" % decode("abcdefgh")
print "%x" % decode("abcdefghi")
print "%x" % decode("abcdefghi")
print "%x" % decode("zbcdeghi")
print "%x" % decode("11111111")
print "%x" % decode("22222222")
```

Output

```
ABab
4377a817de3e
4377a817de3f
4377a817de79
73a9bac7e79d
0
20b1afdbaf7
```

In this program, we have a function called decode. This function converts a Bitcoin address or a string into a long number. Please do not confuse this with the workings of a decode function in the string class.

As before, we create a function called encode. This function takes a long and returns a Bitcoin address as a string. Once again, no connection with the encode function that only deals with strings.

We first decode the string ABab using the decode function. The number returned by the decode function is then encoded using the encode function. Since the values match, it only proves that encode and decode functions work as advertised.

So, with this newly attained confidence, let's pass some larger strings to the decode function. We decode the string abcdefgh and display it. Then the last character h is changed to an i. Please look at the output. Now, the second last digit is changed from g to h, the difference is huge. The biggest change is when the first character a is changed to z. The value changes significantly.

When the bitcoin address is all 1's, the answer is 0. This is because the character 1 is the first character in the base58 string and it has an index of 0. Any number multiplied or divided by 0 is 0. However, when the string is made up of only character 2's, the final number is one digit lesser than the other values.

The characters in a Bitcoin address decide on the weight of a value or a number obtained from the decode function. The numbers vary considerably when characters are closest to the beginning of the string, relatively.

Regenerating a Bitcoin Address from its Hash

```
ch0407.py
from hashlib import sha256
v = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'

def decode(baddr):
    rtot = 0
    for c in baddr:
        rtot = rtot * 58 + v.index(c)
    return rtot

def encode(rtot):
    f = ''
    (ans, r) = divmod(rtot, 58)
    while r:
        f = f + v[r]
        (ans, r) = divmod(ans, 58)
    f = f[::-1]
    return f

baddr = "1BVijayMukhiVijayMukhiVijayMyXUCUJ"
a = decode(baddr)
print "%x" % a
shap = '00731fbe2522335a6a6abad62dd68877714ac76793'
shap = shap.decode('hex')
sha = sha256(sha256(shap).digest()).digest()[:4]
print "sha %s:%d" % (sha.encode('hex'), len(sha))

print encode(int(shap.encode('hex'), 16))
```



```
final = shap.encode('hex') + sha.encode('hex')
print final
print "1" + encode(int(final,16))
```

Output

```
731fbe2522335a6a6abad62dd68877714ac76793025efdd3
sha 025efdd3:4
2c2RRvmwuvPWktBPQCGsd9hPtMBg
00731fbe2522335a6a6abad62dd68877714ac76793025efdd3
1BVijayMukhiVijayMukhiVijayMyXUCUJ
```

The format of a Bitcoin address is :

Byte 1 : version number, Byte 2 – 21 : Bitcoin address, Byte 22-25 : SHA-256 checksum

The raw data of a Bitcoin address has the first byte reserved for a version number. It is followed by 20 bytes which make up, in a sense, the real Bitcoin address. These two fields take up 21 bytes. The end 4 bytes store a truncated SHA-256 checksum. This checksum is the SHA-256 hash of the previous 21 bytes. A Bitcoin address advertised to the public at large is very different from its original form.

To make things easy, we only use the hex notation to display numbers as all numbers between 0 to 255 take up to 2 digits. A byte having a value of 0 is a single digit 0 but it is shown as 00. In the decimal numbering system, the same numbers take up to 3 digits.

So, the internal Bitcoin address format now reads as 2 characters for the version byte, 40 characters for the Bitcoin address and the last 8 characters or digits for the checksum. The total size of a generated Bitcoin address is 50 characters before encoding.

The same encode and decode functions from the earlier code is brought in; the decode function returns a number and not a string.

We will learn how to create vanity bitcoin addresses in the later chapters, but for now, let's look at the bitcoin address 1BVijayMukhiVijayMukhiVijayMyXUCUJ. This is a valid bitcoin address and can be checked in any blockchain explorer. We have concluded a successful Bitcoin transaction with it.

In our bitcoin address, there is no version byte on display. Also, its size is only 34 characters which is nowhere close to the proclaimed 25 bytes or 50 characters. The Bitcoin address are generally either 33 or 34 characters large.

The Bitcoin address is 1BVijayMukhiVijayMukhiVijay and the checksum is MyXUCUJ.

The 1 in the Bitcoin address is problematic as its index offset into the string v is 0. The reason being, it is the first character in the string of valid base58 characters v.

The bitcoin address, when decoded gives a very large number starting with 731. This long number is 48 digits large. The checksum of the bitcoin address is stored in the last 8 digits or 4 bytes.

The variable shap is initialized to this large value manually, with a few additions and subtractions. The digits 00 are appended in the beginning and the last 8 digits of checksum are removed. We could have created this variable by simply appending a '00' to the bitcoin address returned by the decode function, but then we would also have to remove the last 4 bytes. Using hard coded values always helps in our understanding.

We now calculate the checksum of the bitcoin address with the version number included. The net result must match the checksum value shown in the address.

For calculating the checksum, we first decode variable shap and then find its sha256 hash twice. The sha variable

stores the first 4 bytes or 8 digits. After encoding it, you can see that the 8 characters in variable sha match the last 8 digits of the rtot variable, 025efdd3.

Now we regenerate the original Bitcoin address from the value in shap variable. The int function converts a string, the first parameter, into a number. The second parameter tells the int function what base characters the string contains. In our case, it is hex digits or 16. Encoding this number does not result in a correct address. The reason being that the checksum is not joined to it.

So, we first encode the variable shap and then concatenate it with the checksum in the variable sha. (which is only the first 4 bytes). The encode function then encodes it. Finally, 1 is added at the beginning. The output matches the original bitcoin address

Now let's learn how to validate a Bitcoin address.

Validating a Bitcoin Address

```
ch0408.py
from hashlib import sha256
v = '123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'

def decode(baddr):
    rtot = 0
    for c in baddr:
        rtot = rtot * 58 + v.index(c)
    return rtot

def cbaddr(baddr):
    rtot = decode(baddr)
    print "rtot %x" % rtot
    rtotx = '%x' % rtot
    print "Hex %s. Length of the string %d" % (rtotx, len(rtotx))
    print "rtot %d" % long(rtotx, 16)
    zero = len(rtotx) % 2
    print "Zeroes %d" % zero
    rtotz0 = ('0'*(zero) + rtotx)
    print "0's %s" % rtotz0
    final = rtotz0.zfill(25 * 2)
    print "Final %s" % final
    print "Length of Address %d. 0's to be added %d" % (len(rtotx), 50 - len(rtotx) - zero)
    addr = final.decode('hex')
    print "Hex string %d. Decoded hex string %d" % (len(final), len(addr))
    print "Hash %s" % addr[:-4].encode('hex')
    sha = sha256(sha256(addr[:-4]).digest()).digest()
    fhash = sha[:4]
    print "Last 4 bytes %s. First 4 SHA Hash %s" % (addr[-4:].encode('hex'), fhash.encode('hex'))
    match = addr[-4:] == fhash
    return match

ans = cbaddr('1BVijayMukhiVijayMukhiVijayMyXUCUJ')
print ans
print
```

```
ans = cbaddr('1BVijayMukhiVijayMukhiVijayMyXUCUK')
print ans
```

Output

```
rtot 731fbe2522335a6a6abad62dd68877714ac76793025efdd3
Hex 731fbe2522335a6a6abad62dd68877714ac76793025efdd3. Length of the string 48
rtot 2822832147014114415934445864570301114286782606967631183315
Zeroes 0
0's 731fbe2522335a6a6abad62dd68877714ac76793025efdd3
Final 00731fbe2522335a6a6abad62dd68877714ac76793025efdd3
Length of Address 48. 0's to be added 2
Hex string 50. Decoded hex string 25
Hash 00731fbe2522335a6a6abad62dd68877714ac76793
Last 4 bytes 025efdd3. First 4 SHA Hash 025efdd3
True

rtot 731fbe2522335a6a6abad62dd68877714ac76793025efdd4
Hex 731fbe2522335a6a6abad62dd68877714ac76793025efdd4. Length of the string 48
rtot 2822832147014114415934445864570301114286782606967631183316
Zeroes 0
0's 731fbe2522335a6a6abad62dd68877714ac76793025efdd4
Final 00731fbe2522335a6a6abad62dd68877714ac76793025efdd4
Length of Address 48. 0's to be added 2
Hex string 50. Decoded hex string 25
Hash 00731fbe2522335a6a6abad62dd68877714ac76793
Last 4 bytes 025efdd4. First 4 SHA Hash 025efdd3
False
```

The Bitcoin client we downloaded in Chapter 1 can generate multiple Bitcoin addresses.

Any Bitcoin wallet will do the same in a blink. It is very easy to create Bitcoin addresses and compute the checksum.

The bitcoin address in the earlier program, 1BVijayMukhiVijayMukhiVijayMyXUCUJ is a valid address. The website blockchain.info will validate our claim. Simply paste the above address in the text box, and the website will display a list of valid Bitcoin transactions. People pay lots of money to get a car number of their choice; as a very vain person, I like a Bitcoin address with my own name. One can send bitcoins to this Bitcoin address but no one can en-cash them. The reason being that there is no private key to back up this Bitcoin address.

A Bitcoin address must be created from a private key, but we will keep these details for another chapter.

Let's now understand a valid Bitcoin address and ensure the Bitcoin ecosystem accepts it as well. The Bitcoin ecosystem performs two checks mainly, first is the use of a valid Base58 character-set and secondly the checksum.

To summarize, a Bitcoin address is made up of 20 random bytes but it can only contain 58 valid characters. Plus, it has the initial version byte. Every Bitcoin address must end in a valid checksum that is the hash of the earlier 21 characters.

The sha256 hash is 32 bytes large, but we use only the first 4 bytes. If we use the entire hash, the Bitcoin address gets very large.

```
1BVijayMukhiVijayMukhiVijayMyXUCUK.
```

When we enter this second Bitcoin address in the search box of the website blockchain.info, it gets rejected straight

away. The reason is a mismatch in the checksum. We changed the last checksum character from J to K and this change invalidated the checksum.

In the program, as before, we have the string variable `v` that stores the base58 characters allowed in a Bitcoin address. The order of character position in the string must be maintained otherwise websites like `blockchain.info` will not accept the Bitcoin address. Therefore, there is a need to perform a check on these bytes as the Bitcoin system will immediately reject an invalid address.

The function `cbaddr` is called twice, once with a valid bitcoin address and then with an invalid one. Our good old variable `rtot` will contain the sum of the bitcoin characters as a long number. This code is once again encapsulated in the `decode` function.

The variable `rtot` stores a very large number. This long number has 47 or 48 or even 50 digits at times. The large number represents the bitcoin address which is a base58 encoded string. After converting the Bitcoin address to a number, the process can be easily reversed, as learnt earlier.

Try out a few things if you like experimenting. Start replacing each of the characters from the beginning of the string, `baddr` with the digit 2 and notice the value in the `rtot` variable become smaller. Please maintain the length of the bitcoin address at 34. However, if the Bitcoin address is changed to all z's, the length of variable `rtot` touches 49.

Further, we use the `%x` modifier in the print command to convert this very large number into a hex number. Hex values have characters from A to F also wherein A is for 10 and F is for 16. With the `%d` option, every byte occupies 3 digits. A hex number only 2 digits or characters to display a byte. For example, the number 10 in decimal format takes 2 digits, but its hex value A takes the space of one character.

There is no change in the actual value stored in the `rtot` variable. The variable `rtotx` is a string representation in hex format of the value in the variable `rtot`. The `%s` modifier displays it as a string. To the naked eye both look the same, even though one is a string and the other a number. The `int` or `long` function converts the string back to a number.

Only we are aware that the string is a hex representation of a number. But the print function is immune to it and displays the string as it is, without a `0x`.

We are explaining these strange things over and over again because the Bitcoin addresses, which are base58 strings are converted to numbers and then displayed as a string with a hex notation.

The Bitcoin address must have the required minimum size, either 20 bytes or 40 digits depending on the way you like to butter your toast. But it varies. So, we fill this hex string of numbers with leading zeroes. Next, the string must also have an even number of characters and not odd, otherwise the `decode` function will fail.

One of the earlier programs explained the aspects of odd-even size. If the remainder or value returned by the `mod` command is 1 then the string length is odd and we add a leading 0. If the length of the string is even, we add no zeroes. The Python programming language can handle such situations very well.

The length of the string in variable `rtotx` is 48, an even number. The variable `zero` will have a value of 0 since there is no need to fill in any 0s, and the variable `rtotx0` will remain the same as `rtotx`. The `rtotz0` variable concatenates either 1 or 0 zeroes to the hex string `rtotx`.

The length of the string is 48 characters, 40 for the Bitcoin address and 8 for the checksum. The `zfill` string function in Python adds leading number of zeroes to the string. Two zeroes are added to the beginning of the string for the version and make the string 50 digits or 25 bytes large.

It's easier to add to an encoded string than a decoded one.

This string is decoded to obtain the original bytes and calculate the checksum.

The checksum of the Bitcoin address is saved in the last 8 digits of 4 bytes of the string. In the decoded version of the

string, the last 4 bytes of the original string are the actual checksum bytes. This checksum is displayed on the screen. Every Bitcoin address starts with a byte, having a value of 1 or 3. This is then followed by the actual Bitcoin address. The last 4 bytes, 025efdd3 are for checksum.

The double SHA hash value of the first 21 bytes is calculated first and it is stored in the sha variable. Once again, we compute the hash of the string addr, which is the decoded version of string final. The string final has the 0s filled in to get an even number.

The last 4 bytes of the checksum in the Bitcoin address is extracted using the slice operator as in sha[:4]. Only the first 4 bytes of the 32 bytes long SHA hash is looked at and the value is stored in a variable fhash.

These two variables, addr[-4:] and fhash values must match. If they match, then the bitcoin address is well formed and we return true. If it is different, like in the second case, a false value is returned. The SHA hash value and the checksum in the first Bitcoin address have the value of 025efdd3.

The match variable returns false for the second address.

Let's look at the code again. The addr variable contains the decoded Bitcoin address. This includes the version bytes, the actual Bitcoin address or hash and the checksum bytes.

The length of the final variable which is the decoded string is 50 bytes. However, the length of the addr variable is only 25 bytes. It is half in length as it is decoded, and it is padded with 0s to get an even number.

We are displaying the value in the fhash variable, which is the last 4 bytes of the encoded string. The 4 bytes are for checksum and since it is an encoded string, it will be 8 characters.

The decoded string addr is given to the sha hashing function. The string used is the same as seen in the earlier programs. We hash this string and save the entire 32-byte hash in the sha variable. Again, only the first 4 bytes are used as this is the decoded string.

Finally, the last 4 bytes of the addr string are compared with the first 4 bytes of the newly computed sha256 hash which is stored in the fhash variable. It is a convoluted way to validate a Bitcoin address.

Checking a Bitcoin Address with Fewer Error Checks

```
ch0409.py
from hashlib import sha256
v = '123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'

def decode(baddr):
    rtot = 0
    for c in baddr:
        rtot = rtot * 58 + v.index(c)
    return rtot

baddr = '1BVijayMukhiVijayMukhiVijayMyXUCUJ'
print "Bitcoin Address is %s" % baddr

rtot = decode(baddr)
print "Long %x" % rtot
print "Long %s" % rtot
rtotx = '%x' % rtot
print "Hex %s:%d" % (rtotx, len(rtotx))
print type(rtot), type(rtotx)
rtotx0 = rtotx.rjust(25*2, '0')
```

```
print "0 %s:%d" % (rtotx0, len(rtotx0))
rtotx0 = rtotx0 .decode('hex')
sha = sha256(sha256(rtotx0[:-4])).digest().digest()
print sha.encode('hex')
f = sha[:4]
print "%02x %02x %02x %02x" % (ord(f[0]), ord(f[1]), ord(f[2]), ord(f[3]))
g = rtotx0[-4:]
print "%02x %02x %02x %02x" % (ord(g[0]), ord(g[1]), ord(g[2]), ord(g[3]))
print rtotx0[-4:] == f
```

Output

```
Bitcoin Address is 1BVijayMukhiVijayMukhiVijayMyXUCUJ
Long 731fbe2522335a6a6abad62dd68877714ac76793025efdd3
Long 2822832147014114415934445864570301114286782606967631183315
Hex 731fbe2522335a6a6abad62dd68877714ac76793025efdd3:48
<type 'long'> <type 'str'>
0 00731fbe2522335a6a6abad62dd68877714ac76793025efdd3:50
025efdd3c3dee67538e0f42bb260d446bea2df565ac103af18f5534ef97fcf57
02 5e fd d3
02 5e fd d3
True
```

In this program, some error checks are removed, thus it is easier to comprehend. It has fewer lines of code and makes lots of assumptions. We follow the maxim, the fewer the lines of code, the easier to comprehend.

Firstly, the same Bitcoin address of the earlier program is reused and checked to be well formed. The base58 bitcoin address stored in variable `baddr` is converted to a long using the `decode` function.

In Python, a number is displayed using `%x` or `%s` modifiers, the default `%s` uses the decimal notation.

The values stored in the variables `rtot` and `rtotx` are the same but they have a different data type. Python makes it a little difficult to concatenate a digit, 0 to a number, so a string data type is used. The variable `rtotx` and `rtotx0` is the string representation of `rtot` with zeroes added. The actual value remains the same.

The string class has a function, `rjust` to right justify the text. This function is used to add zeroes to the start or the left of the string. This padding makes the string 50 characters large. Some people on the Internet prefer the `zfill` function, others use this approach. We have no view on the issue. The variable `rtotx` is exactly 50 characters long thanks to the `rjust` function. There is no odd-even check performed on the string length.

The sha hash is calculated of the string `rtotx0` and then the hash is encoded for display purpose.

We then extract the initial 4 hash bytes of the entire 32-byte hash display these bytes individually. The variable `f` is used and not `fhsh` and the same method is applied for the variable `g`. The bytes need not be displayed one by one but it works if you have an aversion to a decode or an encode. As before only the first 4 bytes are used for reasons of efficiency.

There are a million methods out there that allow us to validate the checksum. Bitcoin uses a unique way of computing the checksum. It must be noted that because we are using only a tiny part of the hash, there will be collisions.

All in all, a simpler way of explaining the earlier program.

Calculating the checksum using one function from the library

```

ch0410.py
import hashlib
v= '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
v1 = ''.join([chr(x) for x in range(256)])
def chash(s):
    hashtemp = hashlib.sha256(s).digest()
    hashfinal = hashlib.sha256(hashtemp).digest()
    return hashfinal
def decode(baddr):
    rtot = 0
    for c in baddr:
        rtot = rtot * 58 + v.index(c)
    return rtot
def encode(s):
    s1 = ""
    for c in s:
        d = "%02x" % ord(c)
        s1 = s1 + d
    return s1
def encode256(rtot):
    s = ""
    while rtot > 0:
        s = v1[rtot % 256] + s
        rtot = rtot / 256
    return s

baddr = '1BVijayMukhiVijayMukhiVijayMyXUCUJ'
print "Bitcoin Address is %s. Length is %d" % (baddr , len(baddr))
rtot = decode(baddr)
print "rtot %x:%s" % (rtot, type(rtot))
rtotx = encode256(rtot)
print "rtotx %s:%s" % (encode(rtotx), type(rtotx))
print "rtotx %s:%s" % (rtotx, type(rtotx))
rtotx1 = "%x" % rtot
print "rtotx1 %s" % rtotx1
rtotx0 = '\x00' + rtotx
print "rtotx0 %s" % encode(rtotx0)
hash = chash(rtotx0[:-4]):4]
print encode(hash)
if '025efdd3' == encode(hash):
    print "All is well"
else:
    print "Vijay Mukhi is an embecile"

```

Output

```

Bitcoin Address is 1BVijayMukhiVijayMukhiVijayMyXUCUJ. Length is 34
rtot 731fbe2522335a6a6abad62dd68877714ac76793025efdd3:<type 'long'>
rtotx 731fbe2522335a6a6abad62dd68877714ac76793025efdd3:<type 'str'>

```

```
rtotx s?"3Zjj?-wqJ?g?^?:<type 'str'>
rtotx1 731fbe2522335a6a6abad62dd68877714ac76793025efdd3
rtotx0 00731fbe2522335a6a6abad62dd68877714ac76793025efdd3
025efdd3
All is well
```

In this program, we have not used any standard string functions like encode or decode.

The only library function we have used is hashlib.sha256, the one that calculates the SHA hash.

This program works on the same principle as the earlier ones. It validates the checksum of the Bitcoin address. It's like an onion where we peel off one layer after another, replacing one library function at a time, with our own code.

One more variable v1 is added to the code. This variable stores all the 256 characters that a string can literally contain. Most of these characters are unprintable. All the characters in the base58 character-set are hard coded in the program because they are printable and legible.

As explained earlier, in the for loop, the chr function returns a character for every number from 0 to 255 and the join function concatenates them. Eventually, the function join puts all the 255 characters together as a list of valid base256 characters. This is the beauty of python.

Then, in place of performing the double sha hash calculation using standard library functions, we use our old function chash from the file cfuncs.py. The code of this function is embedded in the program.

Further, the variable baddr stores the same old Bitcoin address which has a standard size of 34 bytes. It must be noted that this is only the displayable bitcoin address, the internal structure that generated this address is very different. And, the variable rtot has the computed total of the bitcoin address. The decode function performs the task efficiently, and when the value of the rtot variable is displayed, as before, it starts with 731.

The print function with the modifier %x displays the values in hex format. So, the next task is to convert the value in the rtot variable into a series of hex digits. Since we have ruled out using any Python library code, the function encode256 is created. This function takes a numerical value as a parameter and converts it into a hex string. The return value has the same number.

The function encode256 receives the value of rtot as a parameter. The variable s is an empty string initially but it will contain the hex representation of the value in the variable rtot finally.

The while loop repeats the code until the value of rtot is positive. In the loop, the number in the variable rtot is divided by 256 and not 58. The encoding is base256 and not base58. Since we need the remainder value, the mod operator is used. This remainder is used as an offset into the string v1.

Each character is added at the beginning the string and not at the end, thus sparing us from reversing the string.

Every iteration will reduce the value in the rtot variable and eventually the loop ends.

The output is cross-checked and verified with the library function.

The value in rtotx has many unprintable characters so it's time for some encoding. However, prior to that, is time to add the version byte 0 at the beginning of the string.

Instead of using predefined functions like zfill or rjust, we simply add a \x00 to the string. Remember, the variable rtot is a number, but variable rtotx is a string. The variable rtotx0 when displayed, shows the string beginning with a 00.

The string rtotx has values from the base256 character set. So, to display the value, it must be encoded. So, let's move on to the encode function.

Function encode is first given the decoded string, rtotx and in the next round, it is given the string variable, rtotx0. In the

function, the for loop accesses the individual bytes in the string s. Every character byte is converted into a number using the ord function. The %x modifier gives a hex value but it is in the string format. This string of characters is concatenated to string s1. In the end, the variable s1 stores all these individual characters. The value in the variable s1 is then returned.

The variable rtx0 minus the last 4 bytes is supplied to the chash function to return the sha256 hash of the first 21 bytes of the Bitcoin address. The checksum value is checked with the first 4 bytes of the sha hash value. The slice operator comes in handy here.

The calculated hash value is 025efdd3 in both cases. The values match !!!!

Mission accomplished!!!

CHAPTER 05

Vanity Bitcoin Addresses

This chapter is in continuation of our discussion on bitcoin addresses. Here, the focus is largely on the checksum bytes. We like repeating ourselves, so once again, we discuss the structure of bitcoin address.

The Bitcoin address is 25 bytes large, the first byte is a version byte, then a 20-byte hash and finally 4 bytes for checksum.

The version number is 0. The 20-byte hash is not random but calculated from a public key which in turn is derived from a private key. The public key is hashed using the SHA256 hash algorithm. Unfortunately, the hash size is 256 bits or 32 bytes, a very large number so, a 20 byte RipeMD 160-bit hash is further calculated. That is why, a Bitcoin address is 20-bytes large. The values in the 20 bytes range from 0 to 255. The Bitcoin address always starts with a 0, the initial version byte.

Following the 20 bytes is a 4 byte checksum. These 25 bytes, when encoded into a base58 encoding, becomes a string of 34 bytes in size. The address is then used in Bitcoin outputs.

Enough talk, let's look at some code.

Please enter this Bitcoin address in your favorite blockchain browser like blockchain.info. 1BVijayMukhiVijayMukhiVijayMzzyzz.

The error message says it is an invalid bitcoin address. Or better still, unrecognized search pattern.

Let's write a program that converts this invalid bitcoin address to a valid one. Basically, all that is required is computing a valid checksum. We will be wasting time checking and validating the existing checksum. If we can compute a valid checksum for one Bitcoin address, the same method can then be applied to all other Bitcoin addresses.

Calculating the Checksum of a Bitcoin Address

```
ch0501.py
from hashlib import sha256
v = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'

def decode(baddr):
    rtot = 0
    for c in baddr:
        rtot = rtot * 58 + v.index(c)
    return rtot

def encode(rtot):
    f = ''
    (ans, r) = divmod(rtot, 58)
    while r:
```

```

    f = f + v[r]
    (ans, r) = divmod(ans, 58)
    f = f[:-1]
    return f

baddr = '1BVijayMukhiVijayMukhiVijayMzzyyzz'
print "Length of the Bitcoin address is %d" % len(baddr)
rstr = ""
rtot = decode(baddr)
print "rtot %x" % rtot
rtotx = '%x' % rtot
print "rtotx %s:%d" % (rtotx, len(rtotx))
rtotx0 = ('00' + rtotx).decode('hex')
print "rtotx0 %s:%d" % (rtotx0.encode('hex'), len(rtotx0.encode('hex')))

sha = sha256(sha256(rtotx0[:-4]).digest()).digest()
shap = sha[:4]
shaf = rtotx0[:-4] + shap
print "final %s:%d" % (shaf.encode('hex'), len(shaf.encode('hex')))
print "sha %s:%d" % (sha.encode('hex'), len(sha.encode('hex')))
print "shap %s:%d" % (shap.encode('hex'), len(shap.encode('hex')))

rtotf = int(shaf.encode('hex'), 16)
print "rtot %x" % rtot
print "rtotf %x" % rtotf

baddrf = encode(rtotf)
baddrf = '1' + baddrf
print(baddrf)

if baddrf == '1BVijayMukhiVijayMukhiVijayMyXUCUJ':
    print("All is well")
else:
    print("Vijay Mukhi is an embecile")

```

Output

```

rtot 731fbe2522335a6a6abad62dd68877714ac767933c0d20f3
rtotx 731fbe2522335a6a6abad62dd68877714ac767933c0d20f3:48
rtotx0 00731fbe2522335a6a6abad62dd68877714ac767933c0d20f3:50
final 00731fbe2522335a6a6abad62dd68877714ac76793025efdd3:50
sha 025efdd3c3dee67538e0f42bb260d446bea2df565ac103af18f5534ef97fcf57:64
shap 025efdd3:8
rtot 731fbe2522335a6a6abad62dd68877714ac767933c0d20f3
rtotf 731fbe2522335a6a6abad62dd68877714ac76793025efdd3
1BVijayMukhiVijayMukhiVijayMyXUCUJ
All is well

```

This is the program, which creates our own vanity bitcoin addresses. The addresses can be used to receive bitcoins but they cannot be used to en-cash the bitcoins. The address doubles up as a 'reverse burn bitcoin address' aka a burner

phone. The bitcoins stored with the addresses cannot be recovered as the addresses don't originate from a private key. Even Mr. Nakamoto can do nothing with these Bitcoin addresses. These are called do-not-use Bitcoin addresses.

A valid bitcoin address visible to the human eye normally starts with a 1 and it has a size of 33 or 34 bytes. There is a placeholder like zzyzzz towards the end in place of the checksum bytes. The checksum is not calculated.

Our goal is to compute the SHA checksum. This checksum is verified at every stage with a bitcoin transaction. Every bitcoin address is a base58 encoded string, the checksum is also encoded.

Most of the code in the program is taken from the previous chapter.

The same old variable `rtot` stores the running base58 total of all the bytes including the placeholder checksum bytes using the familiar decode function. This is very important; the value in `rtot` is a total of all the 34 bytes.

As explained in the last chapter, the checksum bytes are placed at the end of the string or to the right end. The left end or the start of the string will not get effected by these bytes. So, if there is a number 326 as a string, on joining 34 to it, this string becomes 32634.

Back to code. The value in the `rtot` variable is converted into hex bytes using `%x` and stored in the variable `rtotx`. The version bytes, a 0 or 00, are added as characters to this hex string. The variable `rtotx0` stores this new value.

As these are early days, we ignore to our own peril the fact that the Bitcoin hex string may be an odd number of bytes. Also, it's too early to think of adding error checks. We could have used `\x00` instead, but not in a mood.

Calculating a hash value of bytes in a string works only with decoded strings. So, the decode function is used to decode the hex string bytes into unprintable numbers. When the bytes are to be displayed, the encode function will come in handy.

The SHA hash of the string variable `rtotx0` is computed and the value is stored in variable `sha`. The last 4 bytes are ignored while computing the hash. Again, only the first 4 bytes of the bitcoin string are used as checksum and the checksum value is stored in the variable `shap`.

Now, to create a new bitcoin address in the variable `shaf`, we use the original hex bitcoin string variable, `rtotx0`. The last 4 bytes, 3c0d20f3, which are the dummy checksum are removed. Instead, we concatenate the hex string with the first 4 bytes of the checksum, 025efdd3, stored in variable `shap`. In effect, we have replaced the original dummy checksum with a new checksum. The values now present in both the variables, `rtot` and `rtotf`, are displayed next to each other to draw comparisons.

The string `shaf` starts with the original Bitcoin hex string which ends in 793. It ends with the checksum bytes 025efdd3. There is the version byte also.

The checksum bytes are 4 bytes large. However, on encoding them using the handy encode function, the size becomes 8 bytes. It must be noted all the time, bytes are encoded always for display purpose.

We now need a base58 encoded string instead of hex digits. So, the good old encode function is used to return a base58 encoded string. The rest of the code remains the same.

In base58 encoding, one byte takes 6 bits and not 8 bits per byte aka base256 encoding. So, the checksum in a bitcoin address takes 6 bytes or 48 bits. In base58 encoding, 8 characters as in $8 * 6$ is 48 bits. One is $8 * 6$, the other is $6 * 8$, the answer is 48 bits. We have explained this mechanism at the end of the chapter.

Finally, to convert the string into a valid bitcoin address, we add 1 as a character type; it has an index of 0 in the string.

Let's add some error checks to make the above program bullet proof.

Adding Error Checks While Computing the Checksum

```

ch0502.py
from hashlib import sha256
import sys
v = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
baddr = '1BVijayMukhiVijayMukhiVijayMzzzzzz'
def decode(baddr):
    rtot = 0
    for c in baddr:
        rtot = rtot * 58 + v.index(c)
    return rtot
def encode(rtot):
    f = ''
    (ans, r) = divmod(rtot, 58)
    while r or ans:
        f = f + v[r]
        (ans, r) = divmod(ans, 58)
    f = f[::-1]
    return f
try:
    for c in baddr:
        dummy = v.index(c)
except:
    print "Invalid Base 58 Character found %c" % c
    sys.exit()
if not(len(baddr) == 33 or len(baddr) == 34):
    sys.exit()
print "Length of the Bitcoin address is %d" % len(baddr)
rstr = ''
rtot = decode(baddr)
print "rtot %x" % rtot
rtotx = '%x' % rtot
print "rtotx %s:%d" % (rtotx, len(rtotx))
zeroes = len(rtotx) % 2
print "Leading 0's are %x" % zeroes
rtotx0 = ('0' * zeroes + rtotx)
rtotx0 = rtotx0.zfill(25 * 2)
print "rtotx0 %s:%d" % (rtotx0, len(rtotx0))
rtotx0 = rtotx0.decode('hex')
sha = sha256(sha256(rtotx0[:-4]).digest()).digest()
shap = sha[:4]
shaf = rtotx0[:-4] + shap
print "final %s:%d" % (shaf.encode('hex'), len(shaf.encode('hex')))
print "sha %s:%d" % (sha.encode('hex'), len(sha.encode('hex')))
print "shap %s:%d" % (shap.encode('hex'), len(shap.encode('hex')))
rtotf = int(shaf.encode('hex'), 16)

```

```
print "rtot %x" % rtot
print "rtotf %x" % rtotf
baddrf = encode(rtotf)
baddrf = '1' + baddrf
print(baddrf)
```

Output

```
Length of the Bitcoin address is 34
rtot 731fbe2522335a6a6abad62dd68877714ac767933c10283f
rtotx 731fbe2522335a6a6abad62dd68877714ac767933c10283f:48
Leading 0's are 0
rtotx0 00731fbe2522335a6a6abad62dd68877714ac767933c10283f:50
final 00731fbe2522335a6a6abad62dd68877714ac76793025efdd3:50
sha 025efdd3c3dee67538e0f42bb260d446bea2df565ac103af18f5534ef97fcf57:64
shap 025efdd3:8
rtot 731fbe2522335a6a6abad62dd68877714ac767933c10283f
rtotf 731fbe2522335a6a6abad62dd68877714ac76793025efdd3
1BVijayMukhiVijayMukhiVijayMyXUCUJ
```

Most programs found on the Internet include too many error checks; they take more exceptions or errors into account. Eventually, the code that checks for errors is larger than the code performing the task.

Our working style is different. We first write code to perform the required task. Once the program has achieved what it set out to do, and if we have spare time on our hands, we build error checks. On the ground, errors and exceptions will have be a part of the program to make it bullet proof but it is placed at a secondary level. This book focusses on the task at hand, it is not to attain skill set for error handling.

We build some error checks in the program. First, the length of the Bitcoin string must always be even. Accordingly, zeroes are added to the string. The program has a variable, zeroes for this purpose.

Second check is on the bytes in the Bitcoin address. We add some extra lines of code to check every character in the Bitcoin address. It should be part of the base58 character set. Thus, the variable c reads every byte of the Bitcoin address stored in variable baddr. The index function throws an exception when it comes across a character not part of the character-set. This situation is likely to occur with characters like 0, O etc. An error message with the invalid character is displayed and the program quits.

The program also quits if the Bitcoin address is not 33 or 34 bytes long. The rest of the code remains the same.

To stress test the program, simply enter the vanity Bitcoin address in the textbox on the blockchain.info website. If the website shows no error, then all is well.

Now let's make slight changes to the bitcoin address in the program.

We initialize the baddr variable with the following bytes.

```
baddr = '1OVijayMukhiVijayMukhiVijayMzzyyyy'
```

Output

```
Invalid Base 58 Character found 0
```

In this case, the offending character is the 0. The program ensures that the Bitcoin address does not contain characters other than a valid base58 encoded character, otherwise it throws an exception and the program ends.

We are not over yet. Let's now stress test the code.

```
baddr = '122222222222222222222222zzzzzz'
```

Output

```
122222222222222222222222yL5ThF
```

Change the value of the Bitcoin address in the variable `baddr` to the value shown above. The program displays the address with a different checksum. Now, paste this Bitcoin address in a browser window running `blockchain.info`. The website marks the new Bitcoin address as valid, it passes the check test, but shows no Bitcoin transactions with it.

For a few addresses now, we keep the last 6 bytes of the checksum fixed as `zzzzzz` and only change the value in the 20 bytes of the RipeMD 160-bit hash.

We choose the number 2 as it has the least weight, accordingly, the `rtot` variable will have the lowest value. In the valid base58 character set, the numbers come first and then the upper case and finally the lower-case alphabets.

Change the value in variable `baddr` to the following.

```
baddr = '1222222222222222222222223zzzzzz'
```

Output

```
1222222222222222222222223wbxbEP
```

The last digit in the series of 2 is replaced with a 3. With this change, it must be noted that first the length is checked to be 33 or 34. The checksum takes up 6 bytes. The rest of the string including the initial character 1 is 28 bytes.

The explanations will come later once we see a few variations in the output.

```
baddr = '1AAAAAAAAAAAAAAAAAAAAAAAAAAzzzzzz'
```

Output

```
1AAAAAAAAAAAAAAAAAAAAAAAAAAwM2D7H
```

Now the digit 2 is replaced with A. It is obvious, that the value in the variable `rtot` will be larger as A has a higher weight or multiplying effect. We pass the test though.

```
baddr = '1WWWWWWWWWWWWWWWWWWWWWWWWWWWWzzzzzz'
```

Output

```
rtotx 143b057e2c13a4b87f3e988c5a208ab3bd4f4fbde7e8fa6ff:49
1WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWX3Ywg6E
```

The website `blockchain.info` gives an error with the above address. One observation to be made here, is on the length of the `rtotx` variable. It is 49 whereas the largest value should be 48.

If the length is 47, we can fill an extra 0 and make it even but we must add 00, 2 digits for the version number. The length becomes 50. We must add the version byte 00 at all cost

There should be an added error check in the program on the length of the `rtotx` variable to bail out gracefully as the value and the length can change significantly.

```
baddr = '1WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWzzzzzz'
```

Output

```
1WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWzz97qr
```



```

    result = v[rtot % 58] + result
    rtot /= 58
    return result

ripe = '56acc14dedbdb502d3e1dd23ca13b4cad4d110bd'
print "ripe %s:%d" % ( ripe , len(ripe))
ripe = ripe.decode('hex')
ripe0 = chr(int(0)) + ripe
print "ripe0x %s:%d " % (ripe0.encode('hex'),len(ripe0.encode('hex')))
print "ripe %d ripe0 %d" % (len(ripe) , len(ripe0))
sha = chash(ripe0)
shap = sha[:4]
final = ripe0 + shap
print "final %s:%d" % (final.encode('hex') , len(final.encode('hex')))
print "shap %s" % shap.encode('hex')
rtot = decode256(final)
print "rtot %x" % rtot
baddr = encode58(rtot)
baddr = '1' + baddr
print "baddr %s:%d" % (baddr, len(baddr))

if baddr == '18uJ4jT1wbVC6SahYGxkTTf35xwQ7MkV66':
    print "All is well"
else:
    print "Vijay Mukhi is an embecile"

```

Output

```

ripe 56acc14dedbdb502d3e1dd23ca13b4cad4d110bd:40
ripe0x 0056acc14dedbdb502d3e1dd23ca13b4cad4d110bd:42
ripe 20 ripe0 21
final 0056acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f:50
shap f3c27a8f
rtot 56acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f
baddr 18uJ4jT1wbVC6SahYGxkTTf35xwQ7MkV66:34
All is well

```

In the previous chapter, we used a private key, 420, to generate a Bitcoin address.

A brief overview of the program.

A library function is used to get the public key from the private key, 420. This public key is hashed using the SHA256 hash and then the RipeMD hash is used to get a 160-bit hash or 20 bytes. The RipeMD hash value of size 40 characters beginning with 56acc is saved in a string variable, ripe.

Since the hash is in encoded form, it is decoded to obtain the original hash value. The decoded hash value is saved in the ripe variable. Then, a version byte, 0, is added to this decoded string in ripe.

The code is polished by using functions that return an int instead of an actual integer value, 0. Thus function int (0) returns the number 0. The chr function joins this number, which is now a character, to a string. The ripped hash is 21 bytes long starting with a 0, the version number.

The variable `ripe` is 20 bytes large and `ripe0` is one larger at 21 bytes. These decoded values can be directly used for calculating the sha256 hash.

The first 4 bytes of the 32-byte checksum is extracted using the slice operator `[:4]` and the variable `shap` holds this checksum value. This checksum is added to the values of `ripe0`. Thus, we have the final Bitcoin address in the variable `final`. It is the version number plus the ripped hash value and the checksum. The final string is 50 bytes large in encoded form or 25 bytes in decoded form.

The word decoding is largely used when we are converting a string into a number. The string variable has only numerical digits but in character form.

The next task is to convert this 25-byte long string into a large number using base 256 encoding. So, applying the same method as before, each character is multiplied by 256.

The `rtot` variable, like before, stores the running total, except that the calculation is spread over multiple lines this time around.

The while loop ends when the string `final` has a length of 0. In the loop, the running total stored in the variable `rtot` is multiplied by 256. Every first character of the string is converted into a number and added to the `rtot` variable. Thereafter, the length of the final variable is constantly reduced by 1 by slicing the first character out. It is similar to base58 encoding except the multiplication factor. The `decode256` function returns the value in the `rtot` variable. The first 40 characters in the `rtot` variable and the RipeMD hash in the `ripe` variable are the same.

The variable `shap` stores the last 8 characters of the newly calculated sha hash. The values in the two variables, `ripe0` and `shap` are then concatenated. This is the final variable.

Now to display this number as a base58 string, we need to encode it. The function to perform this task is called `encode58`, which is just another version of the `encode` function.

In this case, the loop ends when the variable `rtot` is not positive anymore. The variable `rtot` is constantly divided by 58. The remainder is then used as an offset in the `v` string. The character retrieved at this position is concatenated back to back thus giving the final Bitcoin string. Finally, the loop ends when the variable `rtot` has a value of 0, this outcome is possible when the variable `rtot` is divisible by 58. A lot of this is explained earlier.

The digit 1 is added at the beginning of the string to obtain the real bitcoin address. This address matches the hard-coded Bitcoin address we computed in one of the earlier examples.

To sum up, since we are not decoding the ripped hash, the problem of the character 1 having an offset of 0 will not arise in this case. Also, we are simply calculating the hash of the 21 bytes and then concatenating the two decoded values. Further, we have a 20-byte decoded string to work with, the string is not base58 encoded, so there is no need to convert it into a number.

All in all, it is always easier generating addresses with a program than creating our own vanity addresses. But it's good to know the workings and the internal computation also.

In the next example, we replace one more `pybitcointools` function with our own code.

Removing the Library Function Hash160

```
ch0504.py
import pybitcointools
import hashlib
priv = '4200000000000000000000000000000000000000000000000000000000000000'
pub = pybitcointools.privtopub(priv)
pub = pub.decode('hex')
```

```

hash = hashlib.sha256(pub).digest()
print "hash %s:%d" % (hash.encode('hex') , len(hash) )
ripe = hashlib.new('ripemd160', hash ).digest()
print "ripe %s:%d" % (ripe.encode('hex') , len(ripe) )
bitcoinaddress = pybitcointools.hex_to_b58check(ripe.encode('hex'))
print "baddr %s:%d" % (bitcoinaddress,len(bitcoinaddress))
if bitcoinaddress == '18uJ4jT1wbVC6SahYGxkTTf35xwQ7MkV66':
    print "All is well"
else:
    print "Vijay Mukhi is an embecile"

```

Output

```

hash d6645e384af87b5f9dfe838608654ee3efdf3b34f4831e687731c440befb8705:32
ripe 56acc14dedbdb502d3e1dd23ca13b4cad4d110bd:20
baddr 18uJ4jT1wbVC6SahYGxkTTf35xwQ7MkV66:34
All is well

```

In chapter 4, we had the following line in one of the programs,

```
hash = pybitcointools.hash160(pubkey.decode('hex'))
```

In this program, we write only 2 lines of code to perform the same task. Firstly, there is a function sha256, from the module hashlib. This function takes a decoded public key as a parameter and returns a 256 bit SHA hash. Secondly, a new object is created using the function new.

Function new comes from the same library, hashlib, and it creates an object dynamically.

The hash algorithm and the decoded hash of a string are given as parameters to the function. In our specific case, RipeMD160 and the string returned by the previous sha256 function call are supplied. The function returns a 160-bit hash, which is our Bitcoin address. The variable ripe stores this hash value. The function hex_to_b58check called with the encoded hash value as a parameter subsequently returns the actual Bitcoin address.

In this manner, we convert a private key to a Bitcoin address using our code and only four functions from the library. The first function converts a private key to a public key using Elliptical cryptography. The second one computes the SHA hash and finally the last two functions are to compute the RipeMD hash value and the actual base58 address.

By the time we finish this book, we aim to generate a valid Bitcoin address given only a private key of our choosing without using any canned or library functions.

Our First Vanity Bitcoin Address

```

ch0505.py
import pybitcointools
priv = '5KRz2ic53tTk3W7Uv5ZNyFEedCR1yDaKNSLoUimYwJfXgc3Bikj3'
print "priv %s:%d" % (priv , len(priv) )
pub = pybitcointools.privtopub(priv)
ripe = pybitcointools.hash160(pub.decode('hex'))
print "ripe %s:%d" % (ripe , len(ripe) )
baddr = pybitcointools.hex_to_b58check(ripe)
print "baddr %s:%d" % (baddr,len(baddr))

```

Output

priv 5KRz2ic53tTk3W7Uv5ZNyFEdCR1yDaKNSLoUimYwJfXgc3Bikj3:51
ripe 056e7c23675200c9bde27aff07a47ce65cf168b1:40
baddr 1Vijay6huqD74k95N7wnRXGACgaNNG5GY:33
1Vijay6huqD74k95N7wnRXGACgaNNG5GY

The bitcoin address here is backed up with a real private key. We downloaded a program from the Internet to give us a vanity Bitcoin address and then used it to generate real life transactions.

The point here is that only private keys can be used to create real and valid and usable Bitcoin address.

Creating a Bitcoin Address with all 1's

[illegible]

```
Output  
ripe 0000000000000000000000000000000000000000000000000000000000000000:40  
ripe0 0000000000000000000000000000000000000000000000000000000000000000:42  
0:0:0  
zero 21  
sha 94a00911:8  
rtot 94a00911  
final 4oLvT2  
baddr 111111111111111111111111111114oLvT2:27  
All is well  
ripe0 21  
baddr 111111111111111111111111111114oLvT2:27
```

Let's now generate the smallest Bitcoin address which is not 33 or 34 but 27 characters only. The Bitcoin address has all 1's and it is 111111111111111111114oLvT2. This Bitcoin address is a very popular address and the bitcoin.info website shows multiple Bitcoin transactions with this Bitcoin address.

It must be noted that a 0 can be added only to the decoded form of the value. The length of variable ripe0 in encoded form is now 42 characters or 21 bytes.

The next task is to count the number of 0s at the beginning of the string. The `re` library or the regular expression module can be used but we preferred to write our own code in function `getzero`.


```

    }
    output_string.reverse();

```

The repeats at the end is what we need to pay attention to as it talks about leading 0's.

Comparing Different Base Encodings

```

ch0507.py
import math
code_strings = {
    2: '01',
    10: '0123456789',
    16: '0123456789abcdef',
    32: 'abcdefghijklmnopqrstuvwxyz234567',
    58: '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz',
    64: 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+',
    256: ''.join([chr(x) for x in range(256)])
}
def encode(rtot, base):
    bases = code_strings[base]
    result = ""
    while rtot > 0:
        result = bases[rtot % base] + result
        rtot /= base
    return result

def decode(baddr, base):
    bases = code_strings[base]
    rtot = 0
    for c in baddr:
        rtot = rtot * base + bases.index(c)
    return rtot

def sym(base, bytes):
    bits = 8
    logb = 2
    symbols = bytes * bits / (math.log(base, logb))
    return symbols

rtot = 0x56acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f
print "rtot %x" % rtot
baddr2 = encode(rtot, 2)
print "base2 %s:%d" % ('1' + baddr2, len(baddr2))
print "sym2 %f" % sym(2, 25)
baddr10 = encode(rtot, 10)
print "base10 %s:%d" % ('1' + baddr10, len(baddr10))
print "sym10 %f" % sym(10, 25)
baddr16 = encode(rtot, 16)
print "base16 %s:%d" % ('1' + baddr16, len(baddr16))

```

```
print "sym16 %f" % sym(16, 25)
baddr32 = encode(rtot , 32)
print "base32 %s:%d" % ('1' + baddr32, len(baddr32))
print "sym32 %f" % sym(32, 25)
baddr58 = encode(rtot , 58)
print "base58 %s:%d" % ('1' + baddr58, len(baddr58))
print "sym58 %f" % sym(58, 25)
baddr64 = encode(rtot , 64)
print "base64 %s:%d" % ('1' + baddr64, len(baddr64))
print "sym64 %f" % sym(64, 25)
baddr256 = encode(rtot , 256)
print "base256 %s:%d" % ('1' + baddr256, len(baddr256))
print "sym256 %f" % sym(256, 25)
print "base2 %x" % decode(baddr2 , 2)
print "base10 %x" % decode(baddr10 , 10)
print "base16 %x" % decode(baddr16 , 16)
print "base32 %x" % decode(baddr32 , 32)
print "base58 %x" % decode(baddr58 , 58)
print "base64 %x" % decode(baddr64 , 64)
print "base256 %x" % decode(baddr256 , 256)
```

Output

```
rtot 56acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f
base2
11010110101011001100000101001101111011011011110110110101000000101101001111100001
11011101001000111100101000010011101101001100101011010100110100010001000010111101
11110011110000100111101010001111:191
sym2 200.000000
base10 12125260515061556115873075545859999017052292071787046533775:58
sym10 60.205999
base16 156acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f:48
sym16 50.000000
base32 1blkzqkn5w63kawt4hoshstwtfnjuiqxxz4e6up:39
sym32 40.000000
base58 18uJ4jT1wbVC6SahYGxkTTf35xwQ7MkV66:33
sym58 34.141456
base64 1VqzBT29tQLT4d0jyhO0ytTREL3zwnqP:32
sym64 33.333333
base256 1V??M?????#???????z?:24
sym256 25.000000
base2 56acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f
base10 56acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f
base16 56acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f
base32 56acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f
base58 56acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f
base64 56acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f
base256 56acc14dedbdb502d3e1dd23ca13b4cad4d110bdf3c27a8f
```


Before we understand this last program, let's revisit base58 encoding.

Base58 encoding uses 58 characters to encode a single character and only 6 bits are taken. A base256 encoding however, uses 8 bits. This means that 6 bits of the original string take up space of 8 bits. Does this mean that base58 encoding is space inefficient? Now, 2^6 is 64, so we are allowed a maximum of 64 characters to choose from, but the encoding uses only 58 of them. If a string is 2 bytes or 16 bits in base256 encoding or ASCII format, it takes 3 bytes in base58.

The mechanism is as follows: the first 6 bits are read for byte no. 1, then the second 6 bits for byte no. 2 and finally the last 4 bits for byte no. 3. Total number of bytes used are 3. Now, if there are 3 bytes or 24 bits of data to encode, it takes up 4 bytes (24 bits divide by 6 is 4 bytes). Thus, base58 encoding takes up more space compared to base 256.

The SHA checksum is 4 bytes or 32 bits. Converting the value to a base58 string requires 5 reads of 6 bits each, which is 30 bits. The remaining 2 bits takes up only 4 bits out of the next byte. All in all, a checksum in base58 encoding consumes a total of $5 + 1$ (not fully used) = 6 bytes. The SHA checksum takes 4 bytes in base256 encoding but 6 bytes when base58 encoded.

A Bitcoin address takes up 25 bytes, 200 bits. In base58 encoding, it uses 34 bytes with the last byte having unused 4 bits. $34 \times 6 = 204$ bits. This entails a wastage of 4 bits at the end.

On an average, a Bitcoin address is 34 bytes long.

In a hex encoded string of base16, there are only 16 characters to represent a byte and not 58. Only 4 bits are read at a time and not 8. Base16 encoding takes twice the space for a character compared to base256 or normal encoding. A Bitcoin address of 25 bytes will take 50 bytes in base16 encoding, depending upon its length. Base2 encoding is the easiest. Only 1 bit is picked up at a time.

Let's come back to code. It is the best way to understand different forms of encoding and their effects on size. The python libraries, the pybitcointools library and certainly Google provide all the help needed in understanding different concepts.

The program has the same generic encode and decode functions as before but with two changes. Firstly, in base58 encoding, the value is multiplied or divided by 58. And in base256 encoding, it is 256. The encoding format, 58 or 256 is supplied as a parameter to the functions. This number represents the encoding used. The second change is seen in the list of valid characters for every encoding. Every encoding character-set is a subset of the 256 characters.

The bitcoin wiki code displayed above, uses the variable `code_string` to store the valid character-sets. The pybitcointools library also has the same variable `code_strings`. It also uses a python dictionary to store the valid encoding characters. The 'key' used in the dictionary is a pre-defined number for the encoding.

This way the code look cleaner and it avoids the need to have separate variables for every character set.

For the RipeMD 160 hash, the `rtot` variable with the value starting with 0x56 is used. The encode function is given the hash value and the encoding to be used. The function returns an encoded value. Presently, the only encoding which makes sense is the value obtained from the base58 encoding.

Various Bitcoin forums apply some mathematics here. The math says that given a base and length of the Bitcoin address, we can get the number of symbols that the Bitcoin address will need. Thus, if the length of the Bitcoin address is 25, the base 58 encoding will need 34 to 35 symbols. The initial digit, 1 is added to the Bitcoin address after encoding it.

The point to be observed here is that as the base keeps increasing, the number of bytes used also decreases. In conclusion, base2 uses a length of 191 whereas base 10 requires 58 and base58 requires 33 and base256 only 24.

Our understanding of Bitcoin addresses comes from the code of pybitcointools module. For some reason, the above

program threw an exception for the base10 code, the pybitcointools code had an error: 10: '012356789', For some reason, in our copy of pybitcointools, the number 4 went missing.

There are always a few bugs in every code. No code is bug free.

In one of the forthcoming chapters, we will create our own Bitcoin address which will explain the mathematics behind the cryptography used.

Finally, we generate a Bitcoin address with my name in it. To achieve this goal, we downloaded a program called vanity generator from the URL <https://github.com/samr7/vanitygen>. For the mac, we must compile the source code. But prior to the build, replace the first 2 lines of the Makefile file to the following:

```
LIBS= -lpcrc -lcrypto -lm -lpthread
INCPATHS=-I$(shell brew --prefix)/include -I$(shell brew --prefix openssl)/include
LIBPATHS=-L$(shell brew --prefix)/lib -L$(shell brew --prefix openssl)/lib
CFLAGS=-g -O3 -Wall -Qunused-arguments $(INCPATHS) $(LIBPATHS)
./oclvanitygen -d 0 1Vijay
```

Our objective is to see 1Vijay in the Bitcoin address. Depending upon the speed of your Mac it would take from 10 minutes to a couple of hours. The output is shown below.

Output

Pattern: 1Vijay

Address: 1Vijay6huqD74k95N7wnRXGACgaNNG5GY

Privkey: 5KRz2ic53tTk3W7Uv5ZNyFEdCR1yDaKNSLoUimYwJfXgc3Bikj3

The program chooses a random number after executing all the required steps, and generates a Bitcoin address. If the Bitcoin address does not start with 1Vijay, the whole process is restarted. This cycle goes on till it comes up with the bitcoin address we need. Here we also get the private key.

The private key used in the earlier program is generated in this manner. Larger Bitcoin address like 1VijayMukhi could take close to a couple of years to generate a valid Bitcoin address, again with no guarantees.

We could also write a similar program with code used in the previous chapter. The code can be put in a loop to achieve the desired result. The only drawback is that Python would be too slow for our liking.

CHAPTER 06

Difficulty and Nonce

In this chapter, we focus on the last two members of the block header, difficulty and nonce.

Displaying the Last Two Fields of the Bitcoin Header, Difficulty and Nonce

```
ch0601.py
from cfuncs import *
f = open("vijay1.dat" , "rb")
for i in range (0 , 3):
    print "-----Block Number is %d" % i
    rint(f)
    size = rint(f)
    header = f.read(80)
    (ver,prevhash , merklehash , time1 , diff , nonce) =
    struct.unpack("l32s32sll" , header[0:80])
    print "Difficulty or proof of work %x" % diff
    print "Nonce used to calculate the hashes %d" % nonce
    f.seek(size - 80, 1)
```

Output

```
-----Block Number is 0
Difficulty or proof of work 1d00ffff
Nonce used to calculate the hashes 2083236893
-----Block Number is 1
Difficulty or proof of work 1d00ffff
Nonce used to calculate the hashes 2573394689
-----Block Number is 2
Difficulty or proof of work 1d00ffff
Nonce used to calculate the hashes 1639830024
```

The code is taken from one of the previous chapters. Earlier the program displayed all the fields of the block header, here we are displaying only two fields, difficulty and nonce.

Let's first revisit the fields in the block header.

Version number	Hash field	Merkle Hash	Time	Difficulty	Nonce
1	32-byte value	32-byte value	Integer	Integer	Integer

The version number of the bitcoin protocol is always 1, as per the Bitcoin protocol written by Mr. Satoshi. This is followed by the sha256 block hash value of the previous block header. Then is the computed Merkle hash value, which is hash of all the transactions in the block. Following the two hash values is the time of creation of the block. And, towards the end are difficulty and nonce.

For the moment, the difficulty field like the version number, remains the same but for the first 3 blocks only. It changes thereafter and then again remains constant for some time only. The next program explains the difficulty field in greater detail.

Three of the six fields have constant values once computed, whereas the values in other three fields will keep varying. The fields version, prevblockhash and diff have values cast in stone, once computed. They do not change or are not to be changed till infinity. The fields time, merklehash and nonce change periodically.

We set focus on the field called nonce which is the last field in the block header. The value in this field is a very large number, in the billions. So, how does this field get its value?

The answer is based on how a miner mines a block.

A bitcoin miner sets the nonce field to a starting value of 0. Then the sha256 hash of the 6 fields in the block header is calculated twice. The computed SHA hash is a random value and it is 32 bytes large. There will never be two pieces of data that will generate the same hash value, thus, no collisions. Changing the value in the nonce to 1 will generate another random number for the hash value. As the data changes, so does the hash value. It must be noted that this hash value is a very large number.

One of the guidelines in the Bitcoin protocol is that the hash value cannot exceed a certain number. This number is saved in the difficulty field. In case, it exceeds this number then another hash value is computed and the nonce field is incremented by 1. Incrementing the nonce value by 1 generates a new hash altogether. Some data must be altered to compute a new hash value, which ultimately results in changing the value in the nonce field again.

Next, the time field also gets updated intermittently. As a result, a new hash is again calculated. And, the miner also has the option of adding more transactions, thus, changing the value in the Merkle hash field altogether. All in all, each time a new hash is calculated, the nonce value is incremented by 1. This process goes on till the condition is met.

The value in the nonce field for the first block is about 2 billion. It indicates that about 2 billion times the sha hash has been calculated just to arrive at a hash value which is less than a predetermined value, the difficulty. The fundamental concept here is that the number or the value in the difficulty field is not an absolute value.

We can never estimate the time it takes to arrive at a right hash value, therefore, Bitcoin block generation becomes a random event and an expensive one too.

The mining equipment calculating billions of hashes in a blink is no doubt, very expensive. Plus, the running electricity cost in calculating the hash is surely a waste of electricity, thus making it environment unfriendly and inciting the delegates of green peace movement. A lot of Alt coins use a different method, one that is more environment friendly, but none of them are as successful as Bitcoin.

Nevertheless, the transactions in a block, mined by miner A will be very different compared to a block being mined by miner B. No two blocks will ever have the same data.

Our First Stab at Understanding the Difficulty Field

```
ch0602.py
diff = 0x1d00ffff
mant = diff & 0x00ffffff
print "mant %x:%s" % (mant, type(mant))
```

Output

Let's now understand the significance of the difficulty field. The value assigned to this field is generated by the Bitcoin network and not by some miners. It also changes over a period of time, as per the guidelines of the Bitcoin protocol.

One byte is 8 bits i.e. 2 nibbles (1 nibble is 4 bits). A 4-byte value is 32 bits. A byte in hex format takes up 2 characters or digits. So, 1 byte is 2 hex digits/characters and 4 bytes is 8 hex digits or 8 nibbles.

In this program, the variable `diff` represents the difficulty field and it is assigned a value of `0x1d0ffff`. This value is the difficulty level of the first three bitcoin blocks and the block hash must not exceed it.

The diff variable has 8 hex digits and it is to be viewed as two separate numbers, joined together at the hip. The first six hex digits 00ffff are to be extracted separately and then the top two hex digits 1d are treated differently.

To extract these digits, we bitwise and with a 0 or 1. When we bit wise 'and' a bit value with a 0, we are removing or clearing that bit. However, when we bit wise 'and' with a 1 we are keeping the original bit unchanged.

To remove the top 8 bits, the value is bitwise and-ed by 0x00ffff.

Diff	1d	00	ff	ff
&				
00ffff	00	ff	ff	ff
Mant	00	00	ff	ff

The mant variable now has a value 0x00ffff in 3 bytes, 24 bits.

The variable mantx stores the hex value equivalent in mant, but in string format. This helps determine the length of the value. The top byte of the variable diff represents the number of zeroes that the final difficulty must have. Some calculations are made to arrive at the leading zeroes.

Next, to extract the top 8 bits of the diff variable, the bits are right shifted by 24. The last 8 bits is all that is left after the shifting, and this value is stored in a variable, exp.

Diff	1d	00	ff	Ff
>> 24				1d

In this case, the variable exp has a value of 1d or 29 and when doubled, the value is 58.

The value in difficulty is stored across $4^8 = 32$ bytes = 64 hex digits

The difficulty value has leading zeroes and in our case, it is $(64-58) = 6$. The variable zero stores this value.

The total size of the difficulty field is 32 bytes. By choosing the number of leading zeros, we determine the size of the number. Greater the number of leading zeroes, the smaller the final number.

So far, we have arrived at the number of zeroes padded before a value. However, zeroes before a number are generally ignored as 53 is the same as 053 and 0053. Leading zeroes are disregarded.

However, adding trailing zeroes is a different ball game. 53 is very different from 530, 5300 and so on. Trailing zeroes cannot be ignored.

The mantx string gets 6 leading zeroes, thus the intermediate string is 000000 + 00FFFF = 00000000FFFF. The variable tmp contains this value and its length is 12 bytes. The python language has the * operator to repeat a string a certain number of times.

Our final string should be 64 bytes long with hex values. The intermediate string tmp is 12 in size so $(64-12)$ 52 zeroes are added at the end of the string tmp. This 64-digit large hex string is now the value in the difficulty field for the current block hash and the sha256 hash value of the block header should be less than this value. The variable wikix has the same value.

The official Bitcoin Wikipedia adopts a different approach to calculate the difficulty value.

We try and break up the above formula over multiple lines.

As the final variable is a string, the int function is used to convert this hex like string to an actual number. It is a very large number.


```

fffff0000000000000000000000000000000000000000000000000000000000000
000000fffff000000000000000000000000000000000000000000000000000000

```

What we love about programming is that there are multiple ways to skin a cat. You can either obtain the desired result using the vijay mukhi approach (spreading code over multiple line) or you could use `<<` left shifting or `**` based on your mood swings.

The net result is achieving a certain number of zeroes with 1 so that a couple of zeroes can be added at the end.

The leading zeroes are not important and can be displayed using the 064x modifier. Without the modifier, the value displayed is justified to 64 digits, filled up with spaces.

Validating a Block

```

ch0603.py
from cfuncs import *
f = open("/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk00000. dat" , "rb")
#f.seek(20297423, 0)
#f.seek(20300901, 0)
while True:
    where = f.tell()
    rint(f)
    size = rint(f)
    header = f.read(76)
    nonce = rint(f)
    if nonce == 1082:
        break
    f.seek(size - 80, 1)
print 'The file pointer at the start of the block is %d' % where
f.seek(where, 0)
magic= rint(f)
print "Magic Number is %x" % magic
size = rint(f)
header = f.read(76)
nonce = rint(f)
print "Nonce %d" % nonce
print "nonce %08x" % nonce
noncex = struct.pack("<l" , nonce)
print "noncex %s" % noncex.encode('hex')
noncey = struct.pack("<l" , nonce)
print "noncey %s" % noncey.encode('hex')
sha = chash(header + noncex)
print "sha %s" % sha.encode('hex')
(ver,prevhash , merklehash , time1 , diff ) = struct.unpack("I32s32sll" , header[0:76])
exp = diff >> 24
mant = diff & 0xfffffff
target = '064x' % (mant * (1<<(8*(exp - 3))))
targetd = target.decode('hex')
print "target %s:%d" % ( target, len(target))

```

Output

In this example, we validate a block so that it can be added to the blockchain. Simply following the footsteps of the Bitcoin miners!

83

In the first chapter, we had a program that displayed the block header. We reuse the same program here and give it the file, blk00000.dat. Plus, we add an if statement to check the value in the nonce field to be less than 1500. To our surprise, block number 67049 has a nonce whose value is as small as 1082.

This block number on our blockchain starts at an offset of 20300901/20297423 from the start of the file. But why two different numbers?

These numbers surprised us as well. We use a Mac Pro at the office with two monitors and a Retina iMac at home. The block number 67049 was at different offsets in the file blk00000.dat on these two machines,

Nowhere in the bitcoin ecosystem does it say that the bitcoin blocks stored in the blk files have any natural ordering to them. We cannot assume that the blocks are ordered as per creation time or any other column. They all have different data. When you run this program, you will also get a different offset for the above block. In the next chapter, we will give more proof that your blk files are very different than the ones we have.

So, the only way out is to first find the block with the nonce value as 1082. Once the block is located, using the seek function we jump to the relevant position in the file.

As always, in a block, there is first a magic number and then the size of the block data. Thereafter comes the header, of which only 76 bytes are read into a string called header. The last integer (4 bytes) of the header is read into a variable called nonce. Here if the value in nonce is 1082, the keyword break quits out of the loop.

In the file, once the block is found, the file pointer is positioned after the block header. Instead of moving backwards by 88 bytes in the file, the file pointer is repositioned to the start of the block. The values in the where variable aids this repositioning as it is initialized at the start of every block in the loop. The seek function with a value of 0 means an absolute jump from the start of the file.

As before, again the magic number is read and rechecked. It is always better to be safe and not sorry. The header is read minus the last field nonce in the same string variable header and the nonce value is saved in the nonce variable. This technique is not good or bad, but it is our way of writing code.

The value of the nonce variable, when displayed, is 1082.

Thus, to calculate the hash of this block we do not have to change the nonce value 2 billion times, but only 1082 times.

The nonce is a number which is stored on disk as bytes 3a 04 00 00, a little-endian number, but when displayed, it is reversed. The raw bytes in the hex editor will validate our findings. So, 04 is multiplied by 256 to give 1024 and 3a which is 58 in decimal is added to it. $1024 + 58$ is 1082.

To reverse the bytes to 3a 04 00 00, the pack function is used. It takes a number, in our case the variable nonce, and simply reverses the bytes. The value is in string format and stored in a variable, noncey or noncex.

The default format for Intel chips is little endian. A few documents on the web use < sign for little endian, its optional though.

The variables nonce and noncex have the same values for nonce, except that one is the reverse of the other.

The chash function calculates the SHA hash like before. The only difference here is in the structure of the string value. It has two parts. The first constant part is the header which contains 76 bytes of the block header and the second part is the changing value in noncex, in a string format but with the bytes reversed. The variables noncex or noncey either one can be used for this purpose. The value of nonce is changed only when the hash value does not meet the criterion. The 76-byte header remains constant throughout.

With each recurrence, a new nonce value is first reversed, then converted to string format and subsequently concatenated with the header. Thereafter a new hash value is computed. In our specific case, after 1082 iterations, we find a hash lower than the value in the difficulty field.

The variable sha stores the hash of this block and the next line in the program displays it. The next task is to check this value with the difficulty field value. The value of the difficulty field decides if the hash is within the boundaries. This value is very large and it is stored in a variable, target. The difficulty value is calculated before the loop.

The program compares the hash value with the value in difficulty, which is stored in the targetd variable. Remember, both variables, sha and targetd (hex value) are 32 byte strings. We reverse the sha hash when we compare its value with variable targetd.

If the newly minted hash is less than the target difficulty, then and only then the condition will be true. The output indicates that the initial sha256 hash value is much larger than the value in the target difficulty. So back to the loop.

The nonce value is converted into a string with the right endianness, concatenated to the block header and then double sha256 hashed. The if statement is true only when the sha hash is smaller than the target difficulty. Otherwise at the end of the loop, the value of nonce is increased by 1 and the process is repeated with a new hash. The value in the nonce variable is incremented by 1 every time so the hash value will change. The loop ends when condition is met and the values in these variables are displayed.

The variable sharr has a value starting with a 4 and the value in the targetd variable starts with a 5. Obviously, 4 is smaller than 5, though both have the same length. Before the while loop, the length of the sha256 hash variable, sharr was much larger than the target difficulty as a cb is larger than a 5a.

Simultaneously, we converted the string in targetd variable to an encoded value and then computed a number out of it. The miner executes this task in the same fashion, except that that he/she makes billions of computations per block.

Mining the Genesis Block but ...

```
ch0604.py
from cfuncs import *
f = open("vijay1.dat", "rb")
rint(f)
rint(f)
header = f.read(76)
nonce = rint(f)
i = 0
nonce = nonce - 3
(ver,prevhash , merklehash , time1, diff ) = struct.unpack("l32s32sll", header[0:76])
exp = diff >> 24
mant = diff & 0xfffff
target= '%064x' % (mant * (1<<(8*(exp - 3))))
targetd = target.decode('hex')
while i <= 0x100000000:
    noncestr = struct.pack("<l", nonce)
    sha = chash(header + noncestr)
    print sha[::-1] < targetd
    print "nonce%d" % nonce
    print "sharr %x" % int(sha[::-1].encode('hex'), 16)
    print "targetd %x" % int(targetd.encode('hex'), 16)
    print "The value of i is %d" % i
    print
    if sha[::-1] < targetd:
```

```
break
i = i + 1
nonce = nonce + 1
```

Output

False

nonce 2083236890

sharr 36e049f4f81b2952e3f5c8ac41182d36c076fa70c1dbd22a65d39fbe2aa3062e

targetd ffff00

The value of i is 0

False

nonce 2083236891

sharr e21115f96ee0eaad062fd170942b91bb1f43b395757eb87e5a7c97e9823d1383

targetd ffff00

The value of i is 1

False

nonce 2083236892

sharr 8cdc343dc1ca87b0fae7d09a6537a536fec9784a3c4541b770bfb5881c5d66d9

targetd ffff00

The value of i is 2

True

nonce 2083236893

sharr 19d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

targetd ffff00

The value of i is 3

This program calculates the hash of the Genesis block, something we have always wanted to do. At the first attempt, it took over 2 hours so, we located a block with a very small nonce. The file vijay1.dat contains only the first few blocks as our interest is in mining the Genesis block only.

The values of the magic number and size of Bitcoin data are surpassed, and the 76 bytes of the header and the nonce field are saved separately. The nonce value is reduced by 3 and the loop takes over.

In three iterations, we find the hash which meets the condition, wherein the reverse of the sha hash value is less than the value in the difficulty field.

There is really no difference between this program and the last one, except that here we know the block number which has a very large nonce value.

As you can observe from the output, in the first three iterations when i is 0 and 1 and 2, the sha 256 hash value is by far larger than the difficulty.

Only when the nonce is 2083236893, the reverse sha256 hash in variable sharr is smaller than the difficulty. The program works in the same fashion as the last one but it is easier to understand now. Finally, we successfully mined the genesis block and all other blocks in the blockchain. J

A Difficulty with 1a Produces a Smaller Number than a Difficulty with 1c

ch0605.py

def fdiff(diff):

[illegible]

The difficulty bits have three different values starting with 1a, 1b and 1c. These numbers in decimals stand for 26, 27 and 28. The values obtained by multiplying these values, 26, 27, 28 with 2 are 52, 54 and 56 respectively. These values when subtracted from 64 gives 12, 10 and 8. Thus, the first difficulty has 12 zeroes followed by 10 zeroes and 8 zeroes. The greater the number of leading zeroes, the smaller is the number.

Using the Wiki's formula, the value in the difficulty field is converted into a string and then the string is displayed. The count of zeroes match our count of zeroes.

From the output, we infer that the value in difficulty starting with 1a has two more leading zeroes than the one starting with 1b, and hence it is a smaller number. Also, the value in difficulty starting with 1b is smaller than the one starting with 1c as it has two more leading zeroes. The difficulty field with 1c has the least number of leading zeroes so it is the largest number amongst the three.

The int function converts these strings into numbers. As always, the second parameter to the int function is the radix or numbering system to be used.

On printing these long numbers, we can visually see that variable a1c has the largest number of digits and 1a, the least.

A series of if statements re-checks which of the numbers is larger. The output proves that difficulty 1a is the smallest and difficulty 1c is the largest difficulty by value. Just a re-revision.

Double Checking the Fulfillment of the Proof of Work Condition

```
ch0606.py
from cfuncs import *
def fdiff(bits):
    exp = bits >> 24
    mant = bits & 0xffffffff
    return '%064x' % (mant * (1<<(8*(exp - 3))))
no = 0
for fno in range ( 0, 600):
    fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat" % fno
    try:
        f = open(fname , "rb")
    except:
        break
    while ( True ):
        try:
            magic = rint(f)
            if magic == 0:
                continue
        except:
            break
        size = rint(f)
        header = f.read(80)
        (ver,prevhash , merklehash , time1 , diff,nonce) =
        struct.unpack("I32s32sIII" , header[0:80])
        sha = chash(header)
        target = fdiff(diff)
        if sha[: -1] > target:
            print "We have an error in the difficulty block number computation %d:%d" % (fno , no)
            no = no + 1
            f.seek(size - 80, 1)
```

This program generates no output. It is an assurance that all is well and nothing is wrong.

All the Bitcoin transactions are stored in files starting with blk and there are over a 1000 of them. Your mileage will vary. The program attempts to scan every block in every blk file and check on the calculations of the hash. The sha256 hash of every block header is calculated and then checked for it to be less than the difficulty/target. If not, then an error is flagged.

The outer for loop opens each .dat file and it ends at a count of 2000, again a randomly chosen large number. The file name is dynamically generated and the file open command is placed in a try except exception. An exception is thrown when the file with the name does not exist; the loop ends, thus we gracefully quit out.

On the Macintosh, we have a different folder where the files are placed, you will have to change your file name and the root folder as per your Bitcoin folder.

We have no idea how many blocks are present in a single .dat file, so a while loop that loops forever is used.

In this loop, within the try-except block, the magic number or the first 4 bytes are read. If the magic number cannot be read it indicates the end of the file. An exception is thrown, the inner loop ends and gives control to the outer loop for the next file.

If the magic number is 0, the loop starts again where the next 4 bytes are read. The start or end of the block may have 0s.

The size of the data following is stored in a variable called size, it is used in the seek function, which is placed at the end of the while loop. The value is used to position the file pointer to the start of the next block.

As before, the block header is saved in a variable, header and the hash value is computed. Then, the fdiff function is called to obtain the difficulty value from the diff field. Like before, the calculated sha256 hash value of the header is first reversed and then checked with the difficulty string.

The if condition checks if the current block hash is less than the difficulty. If the block hash value is larger than the value in difficulty, it is a grave error and an error message stating the filename and the block number is displayed. Any which ways, the miners pass with flying colors.

The program execution takes some time as there are plenty of blocks to check.

Simple Example to Prove that Blocks are Not Stored on Disk in a Certain Order

```
ch0607.py
from cfuncs import *
import time
def fdiff(bits):
    exp = bits >> 24
    mant = bits & 0xffffffff
    return '%064x' % (mant * (1<<(8*(exp - 3))))
(no , pno , pdiff ) = (0,0,0)
f = open("/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk00416.dat", "rb")
while ( True ):
    try:
        rint(f)
    except:
        break
    size = rint(f)
    header = f.read(80)
    (ver,prevhash , merklehash , time1, diff , nonce) =
```

```
struct.unpack("l32s32sll", header[0:80])
st = time.ctime(time1)
bitcointarget = fdiff(diff)
if diff != pdiff:
    print "Block %03d. Difficulty %x. Time %s. Difference in Blocks %03d" % (no, diff, st, no - pno)
    pno = no
    pdiff = diff
    no = no + 1
f.seek(size - 80, 1)
```

Output

```
Block 000. Difficulty 180a9591. Time Wed Jan 13 11:05:18 2016. Difference in Blocks 000
Block 003. Difficulty 1809b31b. Time Wed Jan 13 12:44:48 2016. Difference in Blocks 003
Block 004. Difficulty 180a9591. Time Wed Jan 13 03:18:10 2016. Difference in Blocks 001
Block 016. Difficulty 1809b31b. Time Wed Jan 13 13:04:15 2016. Difference in Blocks 012
Block 017. Difficulty 180a9591. Time Wed Jan 13 10:09:52 2016. Difference in Blocks 001
```

We have shown only a few lines of output. The program reads only the difficulty field of one block, block number 416. Time and again we have stated that the blocks are not stored in order of the block time created. For example, the output shows how the hours of the block creation time are not in chronological order.

The blocks difficulty changes over time; we need to factor that basic fact. The fact is that the value in the difficulty field changes only once in 2016 blocks.

To ascertain this fact, we figure out the value in the difficulty field of the current block and store it in a variable diff. At the end of the loop, variable, pdiff stores this blocks difficulty value before reading the next block's difficulty value.

If the value in difficulty changes more often, the block data is displayed. Thus, proving that the blocks are not stored on disk date-wise. There are more sophisticated ways to prove this, but we take it up at a later stage in the book.

More Confirmations that there is No Order to the Placement of Blocks in a blk File

```
ch0608.py
import struct
import hashlib

def rint(f):
    return struct.unpack("l", f.read(4))[0]

def computeshahash(s):
    hashtemp = hashlib.sha256(s).digest()
    hashfinal = hashlib.sha256(hashtemp).digest()
    return hashfinal

f = open("/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk00000.dat", "rb")
while True:
    where = f.tell()
    rint(f)
    size = rint(f)
    header = f.read(76)
    nonce = rint(f)
```

```

    if nonce == 1082:
        break
    f.seek(size - 80, 1)
print 'The file pointer at the start of the block is %d' % where
f.seek(where, 0)
j = 0
while ( j <= 3):
    magicnumber = rint(f)
    blocksize = rint(f)
    blockheader = f.read(80)
    (version,prevblockhash , merklehash , creationtime , difficulty, nonce ) =
    struct.unpack("I32s32sIII" , blockheader[0:80])
    hashfinal = computeshahash(blockheader)
    print "Current Block Hash is %s" % hashfinal.encode("hex")
    print "Previous Block Hash is %s" % prevblockhash.encode('hex')
    print
    j = j + 1
    f.seek(blocksize -80 , 1)

```

Output

```

Current Block Hash is
6913fc608acfe605abe19b895bb55402879ca0e49372062b775b080400000000
Previous Block Hash is
ceeb2cd48839879c088e3287daf04391715dbb766f5af24c11e04b0400000000

Current Block Hash is
3a762db2142ed9822a7f1473bcef64773b1c83b20b204f8f89ba330100000000
Previous Block Hash is
ec20f3138ddb2ad2441b3283b1b8e0310599a1ea65dd40748d0f0000000000

Current Block Hash is
ec20f3138ddb2ad2441b3283b1b8e0310599a1ea65dd40748d0f0000000000
Previous Block Hash is
66c17cfab7fcba75cae301b12c52f34e0bef2ef4f3acd8e82b549c0000000000

Current Block Hash is
b394ed0cfe0db8f238a319961abb2ca7d089c4d87ab50b1242cd5e0200000000
Previous Block Hash is
5896f0be9e0137ee9bba97278c888fb67a87d9eb139bb4447ea7d90000000000

```

In one of the earlier chapters, we wrote a program that displays the hash value of the current block and the previous block. The objective then was to explain the concept that the blocks are linked together by the previous block's hash field, the hash value of the previous block.

Let's clarify that stand now. Firstly, the blocks are not stored sequentially in the .dat file. The hash value of each block may not be equal to the hash value of its physical previous block.

The output of the program shows that first block has a hash value starting with 6913. As per our learning, the next block should have this hash value in the previous block hash field but it has a value of starting with ec20. Your mileage here will be different from ours.

The point to be emphasized here is that the blocks are linked to each other by their hashes but these blocks physically may not be placed sequentially in the dat file.

Before we end this chapter, a last look at the concepts of difficulty and the nonce.

$$18\ 24 * 2 = 48\ 64 - 48 = 16\ 64 - 16 = 24$$

$$19\ 25 * 2 = 50\ 64 - 50 = 14\ 64 - 14 = 25$$

$$1a\ 26 * 2 = 52\ 64 - 52 = 12\ 64 - 12 = 26$$

$$1b\ 27 * 2 = 54\ 64 - 54 = 10\ 64 - 10 = 27$$

$$1c\ 28 * 2 = 56\ 64 - 56 = 8\ 64 - 8 = 28$$

$$1d\ 29 * 2 = 58\ 64 - 58 = 6\ 64 - 6 = 29$$

The high byte of the difficulty field indicates the largeness of the difficulty value. Let's start at the largest difficulty value 1d00ffff used in Bitcoin.

1d in decimal is 29, This number is multiplied by 2 to get 58. The total size of the target value is 64 digits. 64-58 gives 6, which is a count of leading zeroes. The same calculation with a hex value of 1c gives 8 leading zeroes. The more the leading zeroes, obviously the smaller the number.

Let's assume, we have a number that uses a grand total of 6 digits, say the number 12. The number 000012 is smaller than 000120 as the number 12 starts with four zeroes and the number 120 starts with only three zeroes.

To conclude, the largest difficulty starts with a value of 1d and then the values simply get smaller.

Let's look at this another way. If there are 6 leading zeroes then the usable digits become 64 - 6 = 58 or 29 bytes for the same value 1d. So, looking at the byte 1d, the number will have 29 digits and difficulty 1c will have a fewer number of digits 28. A number consisting of 29 digits is obviously larger than one that contains 28 or 27 digits.

Assuming the first byte remains constant, then the remaining three bytes determine the targets value. Larger the value, larger is the target, an easy equation.

Now to bring in some clarity. Let's assume the difficulty gives a final target value of 10. Now to mine a block, the hash value must be lower than 10. So, there are only 10 possible hash values to choose from, 0 to 9.

Now let's assume that the target is 100. The hash value can be of a larger horizon now at there can be 100 different values to meet the condition. In other words, it easier to meet the condition by having a larger value.

Thus, a target derived from the difficulty 1d which is a very large number makes it easier to find a sha256 hash value than a target starting with 1c. This is because target 1c has a smaller value than target 1d. In other words, a 1d target is larger than a 1c target. Lowering the target from 1d to 1c takes longer to obtain a sha256 hash as the options decrease. The other point is that the sha256 hash is calculated of the 80 bytes including the last 4 bytes which keep changing relentlessly.

Computing a hash is a very random affair. All values of the nonce and time have the same probability of meeting this condition. So, one can never predict which one will give a hash lower than the target.

The miners may start with a value of 0 for the nonce or a billion, no one can predict the right starting point. Nothing stops a miner from using a random value each time also. Let's not forget the time and/or the Merkel hash. What complicates the matter further, is that we are not looking for a certain hash value but a hash that is smaller than a target value. The chances for 100 miners finding 100 different hashes whose value is less than the target is in the realm of possibility.

The field has rightly been called difficulty because if this is not difficult, then what is!!!!

CHAPTER 07

Storing Bitcoin Transactions using SQL

One of the objectives of this chapter is to understand how Bitcoin determines the difficulty value for a group of 2016 blocks. It undoubtedly, becomes easier if there is some order to the blocks, vis-a-vis creation date and time. We use SQL or Structured Query Language to bring in some order and understand the patterns in the blockchain. We store a part of the Bitcoin data, especially the block header, in a sql database.

Let's start this chapter by running one of the earlier programs. The program displays the previous block hash value and the current block hash value. The creation date of the block is also displayed. This is more to prove that the Bitcoin blocks are not ordered by creation time, when stored on disk. The blocks are not stored using the adage, first come, first served. The placement of these blocks is random. The output you get will be very different from ours as the dat files are not the same on every machine.

Displaying the First Three Blocks from a Block File blk00417.dat

```
ch0701.py
import struct
import hashlib
import time
f = open("/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk00417.dat", "rb")
for i in range (0,3):
    print "Block Number %d" % i
    magic = f.read(4)
    size = f.read(4)
    size = struct.unpack("I" , size)[0]
    header = f.read(80)
    (ver , prevhash , merklehash , time1 , diff , nonce ) =
    struct.unpack("I32s32sIII" , header)
    print "\tPrevious Block Hash is %s" % prevhash.encode('hex')
    shahash = hashlib.sha256(header).digest()
    shahashf = hashlib.sha256(shahash).digest()
    print "\tCurrent Block Hash is %s" % shahashf.encode('hex')
    print "\tTime %s" % time.ctime(time1)
    f.seek(size - 80, 1)
```

Output

Block Number 0

Previous Block Hash is

c927bbd1fe60ef05d0765a908a6621b4156cdac87f4a2b030000000000000000

Current Block Hash is

```
2fe254e01deff1ecc9c90fce9ece0483593bcfde9e9a070200000000000000000000
Time Thu Jan 14 23:29:03 2016
Block Number 1
Previous Block Hash is
4183094969bf4c68545419bc2c89d947dd486a1c8ce94900000000000000000000
Current Block Hash is
d2c593e6de6a9d7468f77ece8b5bd860877529cb923b810700000000000000000000
Time Thu Jan 14 10:05:49 2016
Block Number 2
Previous Block Hash is
3ef6cc78902729ba2c3884de58c5d762bf173576c070d000000000000000000000
Current Block Hash is
aaf8a4d86a3adf945d1d6017b9166da0b6c5a18817d5a10600000000000000000000
Time Thu Jan 14 18:26:12 2016
```

In the program, the first three blocks from file blk00417.dat are read and the relevant field values are displayed. The point to be noted here is that the previous block hash value of block 1 is not the value in the current block hash of block 0. These are named block 0 and block 1 because we are reading only one dat file. Also, the difference in creation time of block 0 and block 1 is in tens of hours. This only proves that the blocks are placed at random in the blk.dat files. As per the Bitcoin protocol, a block is mined every 10 minutes but the date-time stamp showed here has the time difference in hours.

So, let's now proceed to understanding SQL and make things simpler. The good thing about SQL is that the 'L' has nothing to do with programming languages. We will store our data in an open source sql database called PostgreSQL. This database is free and the sql experts believe most professionals prefer it over the others.

We downloaded PostgreSQL from. <https://www.enterprisedb.com/software-downloads-postgres>. There will be a dozen questions to answer while registering. On a Mac, always use the brew installer first to install PostgreSQL. Please do not change any of the default settings.

The program, SQL Shell (psql) is used to run sql commands. Using the program psql, we interact with Bitcoin data and store it in the PostgreSQL database.

A basic lesson in SQL. Skip this part if you know SQL.

Run these statements in psql.

```
create table z1 (fname varchar(100), salary integer);
CREATE TABLE
```

The first SQL command or statement creates a table called z1. 'create table' is part of the SQL syntax, nothing to do with PostgreSQL, and it is given a table-name, z1. The () brackets are part of the SQL syntax. The brackets hold the column or field names of the table, namely fname and salary, and they are separated by commas.

Typically, every column has a column name, i.e. fname, salary etc. and then the columns data type i.e. varchar(100), integer. The column name, fname is for the name of a person and it is a string of characters, therefore varchar datatype. This column is given a maximum size, 100 for the data it would carry, but it is smart enough to use only that much space on disk, as needed.

The second column name is salary and as it is a number, the data type is integer.

The semicolon at the end of the create table statement is a ritual, it is part and parcel of the sql syntax. It works like a full stop.

Now let's add some data to our table z1.

```
insert into z1 values ('vijay' , 10);
INSERT 0 1
```

One way to add data to a sql table is by using the 'insert into' sql statement. This is followed by the table name z1, then the reserved word 'values'. Finally, the () brackets wherein the data or the values are specified. As the column name, fname has a data type of string or varchar, the data is enclosed in single quotes. Numbers need no such protection.

```
insert into z1 values ('vijay' , 30);
INSERT 0 1

insert into z1 values ('mukhi' , 20);
INSERT 0 1
```

After adding three rows or records in the table, let's display this data.

```
postgres=# select * from z1;
 fname | salary
-----+-----
 vijay | 10
 vijay | 30
 mukhi | 20
(3 rows)
```

The select statement is the most common SQL command and it used to get data from the table. The * signifies all fields or column names, followed by the reserved word 'from' and finally the table name.

The 3 rows and 2 columns present in the table z1 are displayed. The select statement with different permutations and combinations can have a book on its own but let's learn just a few basics.

```
postgres=# select * from z1 order by salary;
 fname | salary
-----+-----
 vijay | 10
 mukhi | 20
 vijay | 30
```

The 'order by' clause of the select, orders the data as per a certain column name. By default, the order is Ascending or smallest first. It is also known by the keyword ASC. The opposite keyword is DESC, where the largest value comes first.

```
postgres=# select * from z1 where fname = 'vijay';
 fname | salary
-----+-----
 vijay | 10
 vijay | 30
(2 rows)
```

The 'where' clause is used to select only those rows or records that meet a certain condition. In the above select statement, only those records where fname has a value of vijay is displayed. Remember the single quotes.

```
postgres=# select fname , count(*) from z1 group by fname order by 2;
      fname | count
-----+-----
      mukhi | 1
      vijay | 2
(2 rows)
```

The count (*) counts the number of rows in the result set. The 'group by' clause is a little more complex. It collects all the unique rows that have the same value in the specified column. The field name 'fname' is used in the 'group by' clause.

In table z1, there are only 2 unique rows or sets of values for the field fname. The special keyword count (*) gives a count on the number of rows. Every column in the group by clause, must be present in the select statement output column names otherwise an error is displayed.

The group by clause comes handy when unique values in difficulty are required. It can also disclose the frequency of the difficulty values.

All these SQL statements can also be run using Python. For this purpose, we need to install a module using the import command. The python code can then interact with the PostgreSQL database. We must install a python library, psycopg2.

Python Library Psycopg2

```
ch0702.py
import psycopg2
```

Output

```
ImportError: No module named psycopg2
```

Simply using the import keyword with the module name does not work. There are some zillion modules installed, and the error message indicates that this module psycopg2 is not installed on our machine. So, we use pip to install it, similar to pybitcointools.

```
$ sudo pip install psycopg2
```

Now, the same program shows no error at all. All SQL commands can be executed through a python program.

Run the following command in psql.

```
create table hash1 ( no integer, time1 integer , time1s varchar(128) , diffh varchar(256) ,diff bigint , nonce
bigint );
```

No error messages is a clear indicator that all is well and we are on the right track.

This table hash1 has 6 fields or columns. Most of these fields comprises of the data stored in the Bitcoin block header. The first field is called no and its datatype is integer or number.

This field stores the virtual block number. This field does not exist in the Bitcoin block header as Bitcoin uses a hash to identify a block. It is our own artifact to make life simpler.

Then there are two fields to store the time, one as number and the other in a string format. This is followed by the fields diffh and diff for the difficulty value in varchar type and as a number respectively. And finally, there is the nonce field in big integer format.

Please execute the sql command in psql first. A reminder, the semicolon ends the sql statement, however, it is omitted while writing sql code in python.

The table hash1 is successfully created and data can be entered in it. We will write Python code to insert data in this table.

Connecting to the PostgreSQL database using python code

```
ch0703.py
import psycopg2
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
```

In the Python program, first, a connection must be made to the PostgreSQL database. For this purpose, the function 'connect' from the module psycopg2 is used with the required parameters or a connection string.

The connection string consists the following:

The field dbname is given the name of the database. When PostgreSQL is installed, it creates a default database called postgres. If no specific database is given, then all tables are created in this default database.

Then is the field name called user and it is assigned the default value of postgres.

Thereafter, is the field of password. The value assigned here is the password supplied at time of installing PostgreSQL. Our password is 'accord', your password will be very different. So, you enter your password.

Finally, the field name host is the location of the machine running the database server. The value of localhost means the server and the client code are running on the same machine.

Please bear in mind that all string values are enclosed in single quotes. One more reason why we love Python, strings can be either in single or double quotes.

Inserting into a Table Nearly Half a Million Block Headers

```
ch0704.py
from cfuns import *
import psycopg2
import time
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
no = 0
for fno in range(0 ,60000):
    fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat" % fno
    try:
        f = open(fname , "rb")
        print "Opened File %s" % fname
    except:
        break
    while ( True ):
        try:
            magic = rint(f)
            if magic == 0:
                continue
```

```
except:
    break
size = rint(f)
header = f.read(80)
(ver,prevhash , merklehash , time1, diff, nonce) =
struct.unpack("l32s32slll" , header[0:80])
blkhash = chash(header)
diffs = "%08x" % diff
s1 = "Insert into hash1 values (%d , %d , '%s' , '%s' , %d , %d)" % (no , time1 , time.ctime(time1) ,
diffs , diff , nonce)
print s1
cur.execute(s1)
no = no + 1
f.seek(size -80 , 1)
conn.commit()
print "Number of blocks added %d" % no
```

Output

Opened File /Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk01012.dat
Number of blocks added 487481

This program adds blocks from files, blk00000.dat to blk01012.dat, to the postgres SQL database. After a few years, this number will exceed by leaps and bounds.

After connecting to the database, a cursor object, 'cur' is created.

As always, first, one blk file is opened for reading. If the open command throws an exception, we exit. This is applicable for all the .dat files in the folder.

Then the magic number is read; if its value is 0, we loop back again. After the magic number, the size of the block and the block header are read into the variables size and header respectively.

Once the block header is unpacked into its respective fields, a string variable s1 is initialized to an SQL insert statement. On execution, the s1 string adds a record/row to the table.

The syntax of the sql insert command is 'insert into' then the table name, hash1' followed by 'values' and in round brackets (), the column data separated by commas.

The Sql insert statement generated by our code is given below.

```
Insert into hash1 values (444283 , 1482236095 , 'Tue Dec 20 17:44:55 2016' , '18038b85' , 402885509 ,
3655053615)
```

The python programming language makes the job much easier. The first %d gets replaced by the virtual block number due to Python string substitution's %. The same process is repeated for all the other fields as well.

Once the dynamic string s1 is generated, it must be executed to perform the task. For this purpose, the execute function is used and it comes from the cursor object, cur.

The execute function is not as important as the commit function. If there is no commit, the inserts will get rolled back. The commit function is not part of the cursor object but the connection object, it is the one that adds rows to the table. There is no need to commit after every row or every block. The commit function is placed after reading the entire dat file.

The block number variable is incremented by 1 after the block is added and then the file pointer is positioned to the start of the next block.

Let's manipulate the data in the hash1 table and get the desired results using psql. There may be over 400000 records in the table so to display conditional data, the where clause is used with the sql select command.

We know that the lowest nonce in the blockchain is 1082 and we will prove our findings.

```
select no, nonce from hash1 order by nonce limit 4;
```

```
no | nonce
-----+-----
67061 | 1082
46509 | 1388
117892 | 2048
414595 | 2759
```

The 'order by' clause on the nonce field displays the smallest values of the nonce field first. There may be nearly half a million records in the table hash1 but we are not interested in displaying all of them. A good idea is to limit the number, there will be a faster execution of the select statement and the limit clause is apt for it.

Block no. 67071 has the smallest nonce value, 1082. Once again, these record numbers are deceptive, your block number will vary.

```
select no, time1s from hash1 order by time1 limit 6;
```

```
no | time1s
-----+-----
0 | Sat Jan 3 23:45:05 2009
1 | Fri Jan 9 08:24:25 2009
2 | Fri Jan 9 08:25:44 2009
3 | Fri Jan 9 08:32:53 2009
4 | Fri Jan 9 08:46:28 2009
5 | Fri Jan 9 08:53:48 2009
```

As per the Bitcoin Wikipedia protocol, a new block is added every 10 minutes. The above sql command tries to verify the above statement. The field name 'no' is once again not the block number as the blk files are not stored as per their date of creation. Surprisingly, the time difference between the block numbers 1 and 2 is only a minute, nowhere close to 10 minutes.

The protocol also states that the value in difficulty changes every 2016 blocks. In one hour, the miners would have created 6 blocks. So, in a day, $24 * 6 = 144$ blocks. In 14 days or 2 weeks $144 * 14 = 2016$. Thus, a new difficulty is generated every 2 weeks or 2016 blocks. This is assuming all went well.

How do we check this?

```
select diffh , count(*) from hash1 group by diffh order by 2 limit 3;
```

```
diffh | count
-----+-----
18038b85 | 750
1c0168fd | 2016
1c010c5a | 2016
3 rows)
```

We use the group by clause to group the rows as per the hex representation of the difficulty and count the occurrence of the same.

This code must be executed when there are an adequate number of mined bitcoin blocks. You may have a far greater number of blocks, so it's advisable to use a limit, as in 3 here, but its optional.

```
Select count(distinct diffh) from hash1;  
227
```

There have been 227 different difficulties used in the past. Because of the vast number, we limit the display to only 3. The first difficulty does not have a value of 2016 as the value changes every 2016 blocks. Nearly all the difficulties satisfied the rule and the value changed after a gap of 2016 blocks except for one where the difficulty value of 180375ff was used 2017 times.

```
Select count(diffh) from hash1 where diffh = '180375ff';  
2017
```

```
select count(diffh) from hash1 where diffh = '1d00ffff';  
32256
```

The other value of difficulty, 1d00ffff was used on 32256 blocks and reused 16 times. This value is frequently seen in many blocks. We used the same value in our program as well.

According to the Bitcoin wiki, the Bitcoin ecosystem waits for 2016 blocks to get mined. It then calculates the amount of time taken for them to be mined. If the time taken is longer than 2 weeks, the difficulty value is esteemed to be too high and a smaller value is chosen.

For instance, let's say the last 2016 blocks takes 8 minutes to compute. It indicates that the hash value was being compared with a very large target value. This value must be brought down so that it takes more time for the condition to be met. Conversely, if it is too easy for the miners to obtain the hash, then the difficulty value must be increased so that the time taken for each block to be mined takes more time.

A revision, the hash calculated is checked against a specific target. It must have a value lower than this target. Smaller the target value, the more difficult it is, to pass this condition as there are few values to match the condition. If the target value is larger, then there are more possible values that can meet the condition, hence finding a hash is easier.

Let's now come back to python to determine how often the value in the difficulty field changes.

Understanding Unix Time

```
ch0705.py  
import time  
print time.ctime(0)  
print time.ctime(1)  
print time.ctime(60)  
print time.ctime(61)
```

Output

```
Thu Jan 1 05:30:00 1970  
Thu Jan 1 05:30:01 1970  
Thu Jan 1 05:31:00 1970  
Thu Jan 1 05:31:01 1970
```

Just to understand the concept of time better. All time in Unix and Bitcoin start from Thursday Jan 1st, 1970 at 05:30 hrs. This time is also measured in seconds.

The first output is when the time value is 0, it is the initial value of time. A time of 1 means 1 second from the starting point, the seconds increase by 1. A time of 60 is 1 minute from the start time, 5:30 which now becomes 5:31. A time of 61 means 1 minute and 1 second.

How Often does the Difficulty Really Change, Measured in Days?

```
ch0706.py
from cfuncs import *
import psycopg2
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
cur.execute("select diff,count(*) from hash1 group by diff")
rows = cur.fetchall()
for row in rows:
    #print row[0] , row[1]
    s = "select * from hash1 where diff = %d order by time1" % row[0]
    cur.execute(s)
    rows1 = cur.fetchall()
    #print rows1[0][0] , rows1[0][1] , rows1[0][2] , rows1[0][3]
    #print rows1[2015][0] , rows1[2015][1] , rows1[2015][2] , rows1[2015][3]
    if len(rows1) == 2016:
        tdiff = rows1[2015][1] - rows1[0][1]
        print "diff=%x time=%08d minutes=%05d hours=%03d days=%.02f" % (row[0] , tdiff , tdiff/60 ,
            tdiff/3600 , 1.0 * tdiff/(3600*24) )
```

Output

```
diff=1805a8fa time=01178909 minutes=19648 hours=327 days=13.64
diff=1b01cc26 time=00793357 minutes=13222 hours=220 days=9.18
diff=1c381375 time=00907410 minutes=15123 hours=252 days=10.50
diff=1c05a3f4 time=00289434 minutes=04823 hours=080 days=3.35
diff=1903071f time=00958907 minutes=15981 hours=266 days=11.10
```

The program has the same select statement with the 'group by' clause as before. The execute function also remains the same. We now need some entity by means of which the resultant values of multiple rows and columns can be easily accessed. This entity is called a cursor. The fetchall function returns a tuple of rows and columns and they are accessed using the variable rows.

A tuple is a natural fit for a for statement. The variable row is a tuple and it represents a single row or record. This tuple has the same number of fields as the number of columns in the select statement.

The row [0] element gives the value of the diff column and row [1] has a count of the occurrences of the difficulty. The values returned are accessible as a series of tuples or a set of values. Every iteration of the for statement gives access to an individual row of values. As stated earlier, this diff value is stored in the variable row[0].

To execute another sql statement, the same cursor cur is used and the earlier values are already stored in rows.

The second sql statement retrieves only a selected few values from the hash1 table. These values are from records

where the diff has the same values. The row[0] element has the diff value and it keeps changing in the loop because of the external for statement.

The SQL statement retrieves the records having the same difficulty value. The fetchall function stores them in the variable, rows1. The rows are sorted as per the creation date therefore it shows the initial blocks first. The focus here is on the time difference between the creation time of the first block and the last or the 2016 block.

The tuple rows1 contains all the rows and columns which have the same difficulty value. These rows are also sorted on time. The variable rows1[0] represents the first block. Similarly, the variable rows1[2015] represent the 2016th block.

This field time1 is the second field in the table hash1. Thus, rows1[2015][1] is the time of creation field time1 of the last record. In the same vein, rows1[0][1] is the time of creation of the first block with the same difficulty value.

In the program, we simply minus the time1 field of the last record from the first one. This is because the rows are ordered on time. The variable tdiff stores the difference and it is a value of time in seconds. The value when divided by 60 gives minutes, divided by 3600 gives hours and 3600 * 24 gives days.

The output does not corroborate with the Bitcoin wiki. It takes about 12 to 14 days for the difficulty to change. Nevertheless, we have seen values as low as 3 days. You can print out some values to understand the code better.

Now, we proceed ahead to deal with half a million records. Since the data in the blk files is not stored in an orderly manner, we recreate the hash1 table in the database with all the fields and some extra columns.

So, first drop the earlier hash1 table with the following command.

```
drop table hash1;
```

and then create a new hash1 table with the create command :

```
create table hash1 ( no integer, prevhash varchar(128) , blkhash varchar(128) , time1s varchar(128) ,  
diff integer , nonce bigint , merkle varchar(128) , time1 integer );
```

In the newer version of table hash1, we have included all the fields of the block header. The time field is stored twice, as a number and as a readable string.

Adding all the Fields of the Entire Block Header into a Database

```
ch0707.py  
from cfuns import *  
import psycopg2  
import time  
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")  
cur = conn.cursor()  
no = 0  
for fno in range(0 ,1000):  
    fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat" % fno  
    try:  
        f = open(fname , "rb")  
        print "Opened File %s" % fname  
    except:  
        break  
    while ( True ):  
        try:
```

```

magic = rint(f)
if magic == 0:
    continue
except:
    break
size = rint(f)
header = f.read(80)
(ver,prevhash , merklehash , time1, diff, nonce) = struct.unpack("I32s32sIII" , header[0:80])
blkhash = chash(header)
s1 = "Insert into hash1 values (%d ,'%s' , '%s' ,'%s' , %d , %d , '%s' , '%d')"% (no ,
prevhash[::1].encode('hex') , blkhash[::1].encode('hex') , time.ctime(time1) , diff ,
nonce , merklehash[::1].encode('hex') , time1)
#print s1
cur.execute(s1)
no = no + 1
f.seek(size -80 , 1)
conn.commit()
print "Number of blocks added %d"% no

```

Output

Number of blocks added 485483

This program takes a little longer to execute. Like before, the block headers of all the blocks in the .dat files are read and then inserted into the hash1 table. The table has 8 fields. The hash values are reversed to match the output with sites like blockchain.info

For this program, create a new table, hash2 with the command:

```

create table hash2 ( no integer, pno integer , prevhash varchar(128) , blkhash varchar(128) , time1s
varchar(128) , diff integer , nonce bigint , merkle varchar(128) , time1 integer );

```

This table, hash2 will store all the rows of the hash1 table but in the order of their creation time, as in the time the Bitcoin miners mined the blocks.

The no is the virtual block number and pno is the virtual block number in the hash1 table. This is helpful when there is a need to cross reference with the original row.

Next, create an index on prevhash values in hash1 table.

```

CREATE INDEX vijay ON hash1 (prevhash);

```

To speed up frequent accesses to the tables especially where the volumes are very large, always create an index on the column and use it in the where clause.

Counting the Orphan Blocks in our .blk Files

```

ch0708.py
import psycopg2
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
no = 0
cur = conn.cursor()

```

```
cur.execute("select * from hash1 order by no")
rows = cur.fetchall()
hash = ''
for row in rows:
    blkhash = row[2]
    s = "select * from hash1 where prevhash = '%s'" % blkhash;
    print s
    cur.execute(s)
    row1 = cur.fetchone()
    if row1 is not None:
        hash = row[2]
        break
print hash
cur.close()
cur = conn.cursor()
cur1 = conn.cursor()
cur2 = conn.cursor()
for i in range(0 , 1000000):
    s = "select * from hash1 where prevhash = '%s' limit 2" % hash
    #print s
    cur.execute(s)
    row = cur.fetchall()
    if row is None:
        print "Hell"
        break
    if row is not None:
        s1 = "select * from hash1 where blkhash = '%s' limit 1" % hash
        cur1.execute(s1)
        #print s1
        row2 = cur1.fetchone()
        print "(%04d) %04d %s:%s %s %s %d %s" % (i , row2[0] , row2[1] , row2[2] , row2[3] , row2[4] , row2[5] ,
        row2[6] )
        s2 = "Insert into hash2 values (%d , %d , '%s' , '%s' , '%s' , %d , %d , '%s' , %d)" % \
        (i , row2[0] , row2[1] , row2[2] , row2[3] , row2[4] , row2[5] , row2[6] , row2[7])
        cur2.execute(s2)
        if len(row) == 1:
            hash = row[0][2]
        elif len(row) == 2:
            print "--Orphan Block %s" % hash
            hash = row[1][2]
        elif len(row) == 0:
            print "Over and Out"
            break
conn.commit()
print "Number of records %d" % i
```


Output

```

select * from hash1 where prevhash =
'000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f'
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
—Orphan Block
00000000000000000000e0ab18f7e4811705a573d6523636e3ab1ce9575062305a
—Orphan Block
000000000000000000001f1b9636ac1c6ed6d4c91703510d8e165fa446e7279b3a4
—Orphan Block
00000000000000000000400ca2eb11efc25b0039821d2c6147f1c9b35ddc15e38ab
Over and Out
Number of records 416930

```

This program takes a long time to execute and it displays loads of output. Therefore, we have commented out some of the print statements, you can uncomment them out at your own risk.

After having your tea/coffee, let's run through the program. The first part of the code is redundant, it is more to automate the code as far as possible.

The main objective of this program is to have a cross reference between a hash value of a block and a prevhash value in another block, using sql statements.

The hash starting with 00019 is the hash of block 0. Please remember this hash, as it's the most important hash in the Bitcoin world. All that we are doing is not hardcoding the block 0 hash, which we should have.

To refresh our memory a little bit, in one of the programs earlier, we calculated the block hash, a Sha256 hash, of the current block. Then in succeeding blocks and not the previous blocks, there was a block where the prevhash field in the block header referenced this hash.

If the hash value is not present in any of the prevhash field of other blocks, then the block is called an orphan block. Thus, an orphan block is one whose block hash value is not referenced by any other block's prevhash value. How can this be possible you will ask?

In the Bitcoin ecosystem, the miners try and find a hash that is less than a certain value called the target. This process is random and nobody can predict which miner will generate the value or how long it will take. The nano second the miner solves the above puzzle, he/she broadcasts it to the world. It can happen that another miner at the very same time also solved the sha256 hash puzzle.

Now assume there are two miners, say A and B. Both get different hashes that were lower than the target. Let's further assume that after some time, some other random miner gets a hash for a new block which is lower than the target value. The miner is in a dilemma, as to which block does he/she link this the newly minted block to. Remember, there are two valid blockchains. These two blockchains are identical vis-à-vis all blocks but the last block. This miner would choose the largest chain, but in this case, both chains are the same size. We have two parallel chains, but now one chain is larger as this last block is added to it.

Then 10 minutes later one more block is mined. Here, one chain is larger than the other chain. But due to the inherent lag in networks, the miner may not be aware that there is one chain that is larger than the other one.

Assuming he chooses the larger chain. This chain is now two blocks larger than the smaller parallel chain. At some point in time, more miners will add blocks to this larger chain. Eventually, the smaller chain will die a natural death. This are the vagaries of the Bitcoin protocol. Therefore, there is a wait of 6 blocks to be added over hours to confirm a single transaction. The other blocks just do not die, the transactions they contain will not be thrown away but they are added to the newer subsequent blocks.

At times, miners do not just discard the block from the .dat file. Therefore, there are orphan blocks still lying in the .dat file.

There is no standardization of data in the .dat files, so you may not have the orphan blocks that we have in our copy and so, the outputs may not match.

In 2017, when we executed the same code, we see the following output .

```
select * from hash1 where prevhash =  
'000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f'  
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f  
—Orphan Block  
00000000000000000000000018fc06db522172a181b35cd70c7ffd5fb1e69302af6efe  
Over and Out  
Number of records 447082
```

It is very different from the output we got the first time, in 2016. Your mileage will vary.

```
postgres=# select * from hash1 where prevhash =  
'0000000000000000000000e0ab18f7e4811705a573d6523636e3ab1ce9575062305a';  
no | prevhash | blkhash | time1s | diff | nonce | merkle | time1  
404817 |  
0000000000000000000000e0ab18f7e4811705a573d6523636e3ab1ce9575062305a |  
000000000000000000000010f42545dadcf1262ceebcade6629b16282eb59265c937d | Tue  
Mar 29 15:47:39 2016 | 403088579 | 3284306586 |  
db382ada81f96a5ce47c35c0203d3b3f96d2ff9d71c1ad4350e665b9dcf76566 |  
1459246659  
404818 |  
0000000000000000000000e0ab18f7e4811705a573d6523636e3ab1ce9575062305a |  
000000000000000000000059f3d5b867ffe736a8958273a60846d5b9ea00ba3fb13b7 | Tue  
Mar 29 15:47:49 2016 | 403088579 | 2454628920 |  
8ac753f88fd0e1bd4c0a2a378ae6f5c25cc677b34affb84bf514a7b3961dffffb |  
1459246669  
(2 rows)
```

The sql select statement uses the first hash value. It is our suggestion that you use the first hash of your output and not ours.

Every block header's prevhash field value must reference one and only one block. However, the hash in our program returns 2 rows and not one. This simply means that two blocks in our blockchain have the prevhash value referring to the same block hash value.

In blockchain.info, we write the above block hash starting with e0ab which is the hash value of block number 404811. You can write the above hash in any blockchain explorer and follow along.

We see that the next blocks field on the right of the screen refers to two blocks and not one. Normally the next block has only one hash value. Click on each of the next block hash values. The first hash value is of block number 404812. The second value also takes us to block number 404812 also.

The difference is as follows. The second block link glows in red admitting that it is an orphan block. The first hash says that it belongs to the main chain. These block hashes end with 3b7 and 37d.

Now we run two select statements to determine which prevhash field points to this block hash value.

The hash ending with 37d is an orphan block as there are no blocks linked to it. This is consistent with what we see; the second link is the orphan link. The block numbers however, will not match what you see, reasons explained earlier.

In short, the second select statement checks all blocks for a hash value ending with 26f or starting with 0019 in the previous hash field. This value is the hash of block 0. To verify our claim, enter this hash value in your favorite blockchain browser and you will land up in block 0.

In the program, the variable `hash` is set to the hash value of the first or genesis block. We could have shortened the earlier code by simply assigning it the value of the `blkhash` field of the first row/block 0. Anyways, the basic idea is that the hash value of the field `blkhash` of the first row is checked with the value in `prevhash` field of the remaining rows. Logically, it should be the next row's `prevhash` field, but that may not be the case, all the time.

In the for loop, every record is read in the row variable. Since the field blkhash is the third one in the hash1 table, the variable blkhash is initialized to the value in row[2].

We are not interested in retrieving all the records in the hash1 table as that would be very time consuming. So instead of using the function fetchall, function fetchone is used.

If not true, then we enter the loop again and check the block hash value of the next record. The hash value at the starting point is the genesis hash and it look as follows:

107

Here we have not failed in our attempt to please the purist. That's because we prefer using hard coded values; it simplifies the code you need to understand.

Let's get to the core of the program.

The program has 3 different cursors, cur, cur1 and cur2 as there are three different access to the sql database at the same time. It is important to keep all three result sets separate.

Since we are unaware of the total rows in the table hash1, the for loop variable, i is assigned a very large value. It is like using an infinite while loop though using 'while true' would be a better option any day.

The variable hash has all the records where the value in the prevhash field matches the block hash. Though it is similar to what was done earlier, this select statement fetches a maximum of 2 blocks. If there are orphan blocks, the number of rows returned will be more than 1. The assumption here is that there may be a maximum of one extra orphan per hash.

The limit 2 restriction optimizes the SQL select statement as it does not have to fetch a zillion records. This first sql select statement is in a loop and hence speed and indexes matter.

At the end of the for loop, the value of the hash variable is changed to give a new search string, in other words, the next block hash. Otherwise, we are repeating the same select statement in a loop.

The fetchall function returns a tuple that represents the output of the select statement. A maximum of two rows. The number of rows in the tuple determine if the block is an orphan block or not. If no records are retrieved, the type of row will be None. The if statement is not used for abundant caution. This must not and cannot happen.

There is a block hash referenced by one or more blocks/rows. The row tuple references the block whose prevhash field matches this block hash value. All that is required, is the block header of the block whose hash value is stored in the variable hash.

The first select statement retrieves the block whose prevhash field has the same value as the one present in the variable hash. However, the second sql statement has a condition where it checks for the block whose blkhash value is equal to the hash stored in the variable hash. Both select statements use the same variable's value in the where clause but it checks with different column names, one prevhash and the other blkhash.

The 2nd where clause searches for a row whose blkhash field value is equal to the hash value in the hash variable. The tuple row2, similar to the row variable, has the block header details. The blocks are however different but linked to each other by the hash. These values are added to the hash2 table using the insert statement. It is basically copying all records from the hash1 table meeting some conditions. A different cursor is used for this purpose.

The fetch function fetches a single row from the select statement. Thus, variable row2 comprises only of one row. There is no need for an extra [] operator to access each individual row. The variable i is used as the virtual record number in table hash2 and the variable row2[0] is the row number of the row in table hash1. At some point in time they will have different values.

Now comes the clincher, the steps to take when the number of rows returned in tuple row is only 1 and greater than 1 and less than 1. This condition is checked using the len function. If the value is 1, then there is only one block that references the current block hash. This is the default case. The value of the variable hash is changed using this if statements. Once again, if the size of the tuple rows is 1, then a certain field off this tuple is extracted. The hash variable's new value is the third field of this row, i.e. blkhash lest you have forgotten.

Now if the length of the tuple row is 2, then there is an orphan block. It is assumed that the second row is the orphan block and it has a hash value. The actual orphan block can be identified by writing more SQL statements code.

Finally, if the tuple length is 0, then it is assumed, and rightly so, that no rows are returned. This can only happen when

we have reached the last Bitcoin block in the blockchain. This block obviously has no blocks whose prevhash field value refers to it. It is the top most block in the blockchain. The highest block in the chain will have no block referring to it.

The first and last prevhash fields must be handled with care. The first prevhash field has a value of all 0's.

The commit function ensures that all data gets added to the hash2 table.

Two of the orphan blocks from the previous output are

—Orphan Block

00000000000000000000e0ab18f7e4811705a573d6523636e3ab1ce9575062305a

—Orphan Block

000000000000000000001f1b9636ac1c6ed6d4c91703510d8e165fa446e7279b3a4

postgres=# select * from hash1 where prevhash =

'00000000000000000000e0ab18f7e4811705a573d6523636e3ab1ce9575062305a';

no | prevhash | blkhash | time1s | diff | nonce | merkle | time1

404818 | 00000000000000000000e0ab18f7e4811705a573d6523636e3ab1ce9575062305a |

0000000000000000000059f3d5b867ffe736a8958273a60846d5b9ea00ba3fb13b7 | Tue

Mar 29 15:47:49 2016 | 403088579 | 2454628920 |

8ac753f88fd0e1bd4c0a2a378ae6f5c25cc677b34affb84bf514a7b3961dffffb |

1459246669

404817 |

00000000000000000000e0ab18f7e4811705a573d6523636e3ab1ce9575062305a |

0000000000000000000010f42545dadcf1262ceebcade6629b16282eb59265c937d | Tue

Mar 29 15:47:39 2016 | 403088579 | 3284306586 |

db382ada81f96a5ce47c35c0203d3b3f96d2ff9d71c1ad4350e665b9dcf76566 |

1459246659

(2 rows)

postgres=# select * from hash2 where prevhash =

'00000000000000000000e0ab18f7e4811705a573d6523636e3ab1ce9575062305a';

no | pno | prevhash | blkhash | time1s | diff | nonce | merkle | time1

404812 | 404818 |

00000000000000000000e0ab18f7e4811705a573d6523636e3ab1ce9575062305a |

0000000000000000000059f3d5b867ffe736a8958273a60846d5b9ea00ba3fb13b7 | Tue

Mar 29 15:47:49 2016 | 403088579 | 2454628920 |

8ac753f88fd0e1bd4c0a2a378ae6f5c25cc677b34affb84bf514a7b3961dffffb |

1459246669

(1 row)

The same sql statements on both the tables return two rows for hash1 and one row for hash2. The orphan blocks from table hash2 have been eliminated.

Another way of figuring out the number of orphan records is as follows.

postgres=# select prevhash, count(*) from hash1 group by prevhash having count(*) > 1;

prevhash | count

00000000000000000000e0ab18f7e4811705a573d6523636e3ab1ce9575062305a | 2

000000000000000000001f1b9636ac1c6ed6d4c91703510d8e165fa446e7279b3a4 | 2

(14 rows)

Showing only part of the output.

This sql statement takes a little longer to work its magic.

We use the same group by clause on the field prevhash. This will group together, all records with the same previous block hash values. The 'having' clause works only with the group by and it is a condition. Thus, only those previous block hashes that have more than one value will satisfy the condition. These are the hashes of the orphan blocks, only one hash of the two is valid.

```
postgres=# select prevhash, count(*) from hash1 group by prevhash having count(*) > 1;
prevhash | count
-----+-----
00000000000000000000000018fc06db522172a181b35cd70c7ffd5fb1e69302af6efe | 2
(1 row)
```

Run the same select statement on the data as in

```
postgres=# select prevhash, count(*) from hash2 group by prevhash having count(*) > 1;
prevhash | count
-----+-----
(0 rows)
```

Remember table hash2 has no orphan blocks.

Checking the integrity of the blocks in the hash2 table

```
ch0709.py
import psycopg2
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
no = 0
cur = conn.cursor()
cur.execute("select * from hash2 order by no")
rows = cur.fetchall()
for row in rows:
    #print row[0] , row[2] , row[3]
    if row[0] == 0:
        hash = row[3]
        continue
    if row[2] == hash:
        #print "Row %d passed" % row[0]
        pass
    else:
        print "\t\tWe have an error at row %d" % row[0]
        hash = row[3]
```

The select command retrieves all the records from the hash2 table, ordered by no. This will bring in some consistency as we have no idea how the rows will be returned. Then a loop construct is placed to iterate through each row. The genesis block or row 0 is handled with kids gloves. This genesis block is not a valid block as it is never mined. It exists only in C++ code. The first if statement checks if it is block 0/genesis block.

The variable hash is assigned the sha256 hash value of the block. It can get the value either from row[3], the fourth

Output

```
18059ba0
000000000000000059ba0000000000000000000000000000000000000000000000000
59ba0000000000000000000000000000000000000000000000000000000000000000
ffff0000000000000000000000000000000000000000000000000000000000000000
196061423939.65
```

The difficulty value of the block 416374 in decimal format has a value of 403020704, though they are best seen in hex.

The function `fdiff` converts the difficulty into a target value to compare it with the sha256 hash value. Hence, the leading 0's. The `int` function convert this hex string into an actual number in the variable `diffx`. All the leading zeroes magically disappear.

Now we repeat the same process all over again but with the difficulty value `0x1d00ffff`; it results in the highest possible target value. The value is stored in variable `diffx1`.

The largest target value in `diffx` is divided by the value in the difficulty of block 416374. This resultant value when displayed, is the same as shown in the `blockchain.info` difficulty field for block 416374.

The output displayed is a ratio, also called `bdiff`. It indicates that the current difficulty is a very small value. Thus, it would be 196061423939.65 times harder to meet the hash condition with this difficulty of block 416374 than compared to the difficulty `1d00ffff` of the initial blocks.

Let's divide a random number 30 by 10. 30 and 10 are assumed to be target values.

This division results in 3, i.e. the numerator is 3 times larger than the denominator. In other words, it is 3 times more difficult to find a hash value less than 10 compared to 30.

We can denote the maximum difficulty `0x1d00ffff` as 1. This value gives the highest possible target value that can be used in the Bitcoin ecosystem.

Please remember that any difficulty value remains constant only for 2 weeks.

Pool Difficulty or `pdiff`

```
ch0711.py
a = 0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
b = 0x00000000FFFF0000000000000000000000000000000000000000000000000000000
print a - b

a = 0x00000000FFFF0000000000000000000000000000000000000000000000000000000
b = 0x000000000000404CB0000000000000000000000000000000000000000000000000000
c = 1.0 * a / b
print "%f" % c
d = a / b
print "%d" % d

a = 0x00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
b = 0x000000000000404CB0000000000000000000000000000000000000000000000000000
c = 1.0 * a / b
print "%f" % c
print "%d" % c
```


We call ourselves very experienced Google search experts. We reached every nook and cranny of the Internet to search for Bitcoin concepts like pdiff, bdiff and we came a cropper. Everyone redirected us back to the same Bitcoin wiki page on difficulty. But we did come across one group paper from the University of Rutgers that said more than the Bitcoin wikipedia. The above program is taken verbatim from them.

Let's get back to the Bitcoin wiki to help us compute the total computing capability present in the Bitcoin network.

113

Normally, the value in difficulty is not 1, so let's say it is D. For difficulty D, the offset would be

$$(0xffff * 2^{208}) / D.$$

The expected number of hashes to find a block of difficulty D is (the wiki's words not ours)

$$D * 2^{256} / (0xffff * 2^{208})$$

The Bitcoin wiki does not like dealing with large numbers. There is no point multiplying and dividing by large numbers when the common factor for both is 2^{208} . If this common factor is removed from both the numerator and denominator, we get.

$$D * 2^{48} / 0xffff$$

This is because we are subtracting 256 from 208, it gives a value of 48. We are calculating $D * 2^{48} / 0xffff$ hashes in 600 seconds.

$$D * 2^{48} / 0xffff / 600$$

We can simply compute this over 2016 blocks to give us

$$D * 2^{32} / 600.$$

This is an approximation that we used.

Processing Power of the Bitcoin Network then and Now

```
ch0714.py
def fdiff(diff):
    exp = diff >> 24
    mant = diff & 0xfffffff
    target = '%064x' % (mant * (1 << (8 * (exp - 3))))
    return target

import psycopg2
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
no = 0
cur = conn.cursor()
cur.execute("select count(*) from hash2")
row = cur.fetchone()
tot = row[0]
cnt = tot / 2016
print "No of loops %d:%d" % (tot, cnt)
cur.execute("select * from hash2 order by no")
rows = cur.fetchall()
index = 0
pghz = 0
prevdiff = int(fdiff(0x1d00ffff), 16)
for i in range(0, cnt):
    if rows[index][5] != rows[index+2015][5]:
        print "The first and last difficulties do not match"
        break
```

```
tdiff = rows[index+2015][8] - rows[index][8]
days = tdiff / (60*60.0 * 24)
diffm = fdiff(0x1d00ffff)
diffm = int(diffm , 16)
diffx = int(fdiff(rows[index+2015][5]),16)
diffs = 1.0 * diffm / diffx
ghz = diffs * 2 ** 32 / 600
ghzp = (ghz * 100.0) / 7158278
print "diff=%x secs %7d days %05.2f %12d %20d %17.0f" % (rows[index][5] , tdiff , days , diffs , ghz , ghzp)
index = index + 2016
```

Output

```
No of loops 447083:221
diff=1806274b secs 1209731 days 14.00 178678307671 1279029146590879744 17867832830618
diff=18062776 secs 1112491 days 12.88 178659257772 1278892782101064192 17865927840482
diff=1805a8fa secs 1178909 days 13.64 194254820283 1390530167012915968 19425484271677
diff=18058436 secs 1229655 days 14.23 199312067531 1426731352908053760 19931209054860
```

This may sound insane but in the above program, we decipher the processing capacity of the Bitcoin network, from the early days to the day of this program's execution.

Again, the number of records in the hash2 table will be a large number as it contains the entire history of all the Bitcoin transactions.

The first select statement is executed with the clause count (*). It returns only 1 row and 1 column. Here the function fetchone is used to fetch the single row. The total number of records in the hash2 table is insignificant but what's more important is the count of different values in difficulty. This number fortunately changes only once every 2016 rows and this count is stored in the variable tot.

The value in tot is divided by the magic value 2016. This helps loop over every group of similar difficulties. The cnt variable is used later in the for loop. For the record, the loop is executed 221 times, your mileage will obviously be greater.

The next select statement takes some time to execute as it is busy retrieving nearly half a million rows using the function fetchall. Plus, all columns are also retrieved, though not needed. The order by clause makes sure things work as advertised. The rows tuple stores all our block header data.

The for loop is executed 221 times. There are 221 * 2 or 442 weeks of Bitcoin transactions. The difficulty changes every 2 weeks. Dividing by 52 gives approximately 7 years of transactions. The first transaction is dated Jan 2009 and the last depends on when we run our code.

At first, we have some error checks. Since the value in difficulty remains the same for a set of 2016 consecutive blocks, the value of the first and the last block in a set are checked to be the same.

The rows variable is nothing but a list. The size of the list is the number of rows and each row stores the column data, the block header fields. The fifth column contains the difficulty value. In the if statement, the values of relative row numbers 0 and row numbers 2015 are checked. Since the index variable starts with a value of 0, rows 0 and 2015 are checked. In the next round, the value of the index variable is increased by 2016. So, rows 2016 to 4031 will be checked. If the value in difficulty of the rows within the loop iterations do not have the same value, an error message is printed and the loop ends. Thankfully, we do not see any error messages in the beginning. However, towards the end, the last rows may not have the difficulty value spaced out between 2016 rows as there may not be enough rows.

The 8th column or last column has the time in seconds. This is the block creation time. It is a time relative to a certain day in history.

To find the difference between the largest and smallest block time creation, we subtract the value in 8th column of row 2015 from the 0th row. The difference is saved in the variable `tdiff`. The 2015 block will have a larger time value than the first or 0th block, relatively.

Since the time is in seconds, the difference in days is achieved by converting the value to days first and then in hours and minutes. This is achieved by dividing the value with the outcome of 3600 multiplied by 24. The variable `days` stores this converted time value.

The next task is to calculate the difficulty. The variable `diffm` stores a string representation of the difficulty value of 1 or 0x1d00ffff. The `fdiff` function is reused as always.

Since the value is in a string format, it cannot be used directly. The `int` function convert this string with hex values into a real number. This variable is also called `diffm`.

Further, the difficulty value, column 5 of the 2015th row is also converted to an integer. The value is saved in variable `diffx`. To arrive at the actual value of difficulty, the value in `diffm` is divided by the one in `diffx`. The `diffs` variable stores the result. The multiplication by 1.0 is to get a decimal value in the answer.

For the processing capacity of the Bitcoin network, we simply multiply the difficulty value in variable `diffs` by 2^{32} and then divide by 600 seconds as explained earlier. Variable `ghz` is the processing power capacity.

Finally, the values in the difficulty, seconds, days, difficulty and processing capacity are displayed

The output can run into pages, so we simply look at the last rows. The last processing power is 19 digits' large number, it simply keeps growing. The percentage of increase of the processing power over the years is measured by dividing the value with 7 Ghz, the initial processing power of the Bitcoin mining network. This result is saved in the variable `ghzp`.

To check the veracity of our code, we used the example from the Bitcoin wiki where a difficulty of 22012 had a processing power of 157 GHz. Then difficulty 0x1b0404cb had a `bdiff` of 16307.

A few things that we learnt here is that in the `%f` format, the first number is total size of the field including the decimal point and the decimal places. Programming languages look at formatting options differently.

Percentage Difficulty Change

```
ch0715.py
def fdiff(diff):
    exp = diff >> 24
    mant = diff & 0xffffffff
    target = '%064x' % (mant * (1<<(8*(exp - 3))))
    return target

daysp = (7 * 100) / 14.0
print "%5.2f" % daysp
daysp = (28 * 100) / 14.0
print "%5.2f" % daysp
diff1 = int(fdiff(0x1806274b) ,16)
diff2 = int(fdiff(0x18062776) ,16)
diffp = (diff2 * 100.0) / diff1
print "%7.3f" % diffp
diffp = (diff1 * 100.0) / diff2
```

```
print "%7.3f" % diffp
```

Output

```
50.00
200.00
100.011
99.989
```

This is a lull before the storm. Let's assume that 2016 blocks get mined in 7 days and not 14, so, it is assumed the miners have received their mining reward 50% earlier. The calculation followed is :

It should take 14 days to mine, but it took 7

It should take 100 days to mine, it will take ?

$100 * 7 / 14.0$ is the equation.

To calculate the percentage increase or decrease in the number of days it takes from the base of 14 needs only one variable. The problem comes with the difficulty. The value in difficulty must be increased or decreased by the same amount as per the days overshoot beyond 14.

Let's take up the case of 2 values in difficulty seen in the earlier program, 0x1806274b and 0x18062776. The million-dollar question is, what is the percentage increase between the two difficulties.

The value in the difficulty field represents a large number. In its present form, they are of no use. So, we use our good old diff function to convert them into real usable values. It must be noted that unlike time, here the comparison is between the current value of a row with its previous value. They are two different values from two different rows. In a sense, the current value of the difficulty is checked with the value in the difficulty field of the previous row.

Prev difficulty was 1806274b, new is 18062776

Prev difficulty was 100, new is

$100 * 18062776 / 1806274b$

The values when reversed give a wrong answer. The percentage increase is 100.011 percent.

Making Sure that the Difficulty Value Changes by the Right Amount

ch0716.py

```
def fdiff(diff):
    exp = diff >> 24
    mant = diff & 0xffffffff
    target = '%064x' % (mant * (1 << (8 * (exp - 3))))
    return target

import psycopg2
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
no = 0
cur = conn.cursor()
cur.execute("select count(*) from hash2")
row = cur.fetchone()
cnt1 = row[0]
cnt = cnt1 / 2016
```

```

print "No of loops %d:%d" % (cnt1, cnt)
cur.execute("select * from hash2 order by no")
rows = cur.fetchall()
index = 0
prevdiff = int(fdiff(0x1d00ffff) , 16)
prevdays = 0
prevper = 0
pdaysp = 0
for i in range ( 0 , cnt):
    if rows[index][5] != rows[index+2015][5]:
        print "The first and last difficulties do not match"
        break
    tdiff = rows[index+2015][8] - rows[index][8]
    days = tdiff/(60*60.0 * 24)
    daysp = (days * 100) / 14.0
    diffx = int(fdiff(rows[index+2015][5]),16)
    diffp = (diffx * 100.0) / prevdiff
    print "diff=%x days %06.3f percent %07.3f:%07.3f %07.3f "\
    % (rows[index][5] , days , daysp , diffp , pdaysp )
    prevdiff = diffx
    pdaysp = daysp
    index = index + 2016

```

Output

```

diff=1806274b days 14.002 percent 100.011:093.381 093.381
diff=18062776 days 12.876 percent 091.972:100.011 100.011
diff=1805a8fa days 13.645 percent 097.463:091.972 091.972
diff=18058436 days 14.232 percent 101.658:097.463 097.463

```

Most of the code remains the same as before. The variable `daysp` stores the percentage difference from 14 days, either positive or negative. In theory, the difficulty changes every 14 days. As per the output displayed, whenever the variable `days` value is greater than 14, the percentage displayed is also larger than 100. Where the `days` variable has a value of less than 14, the percentage value displayed is also less than 100. The row where the value of the `days` variable is 10.589 days, it has a percentage displayed to the right of 75%.

The real calculation is in the difficulty.

If the `days` percentage has a value of 75%, it indicates that the 2016 blocks got mined before time. Therefore, the time taken to mine the blocks must be increased. This can only happen if the target value is lowered, thus making it more tedious for the miners and eventually slowing the mining process.

The initial blocks with the difficulty value of 0x1d00ffff are ignored.

Let's take the following case.

```

diff=1803d589 days 12.944 percent 092.459:098.269 098.269
diff=18038b85 days 13.668 percent 097.628:092.459 092.459

```

It takes less than 14 days or 12.944 days to mine 2016 blocks. The current difficulty is 1803d589, so it is reduced to 18038b85. Thus, if the mining is faster, the difficulty value of the next set of blocks is reduced

```
diff=18052669 days 14.804 percent 105.745:099.956 099.956
diff=18057228 days 13.003 percent 092.878:105.745 105.745
```

In this case, it takes 14.804 days to mine the blocks, too much time. The current difficulty was 18052669, so the next difficulty value chosen was 18057228, the value in difficulty is increased. It's the opposite of the earlier case.

The current target value stored in variable `diffx` is multiplied by 100. The value is the last difficulty value of the 2016 block set. The resultant value is then divided by the previous difficulty, the variable `prevdiff` or the difficulty used by the previous 2016 blocks. Programming wise, at the end of the loop, we simply store the previous difficulty in a variable `prevdiff`. To crosscheck, the value of the percentage increase in days of the current difficulty set must match the percentage increase of the difficulty of the next block's last two columns.

```
diff=1804de5e days 13.107 percent 093.618:097.752 097.752
diff=18048ed4 days 13.063 percent 093.310:093.618 093.618
```

The percent column of the earlier difficulty is 093.618, the next line shows the last two columns are 093.618.

The previous value of the percentage difference of days is displayed side by side to cross verify. Comparing floating point numbers is a dying art. The last 2 columns must match and they do.

All that is proved here, is that the Bitcoin code increases or decreases the difficulty value depending upon the difference in time taken for the last 2016 blocks to be mined.

Let's explain this again.

```
diff=180440c4 days 14.271 daysp 101.934:093.310 093.310
diff=180455d2 days 13.945 daysp 099.607:101.934 101.934
diff=18045174 days 12.650 daysp 090.355:099.606 099.607
```

Let's start with a clean slate.

When the miners took more than 14 days to mine 2016 blocks, it was time to make things easier for them. So, the value in difficulty was changed from 180440c4 to 180455d2. The increase in the difficulty value gave them more sha hashes to meet the condition on paper at least and thus reduced the time.

The last two columns in the output have the same value in the second line, 101.934. The second last column is the difficulty percentage and the last column is the previous days percentage and not the current days percentage.

The current days percentage, `daysp`, reveals if the time taken has exceeded the 14 days or not. If yes, then the value is above 100%, otherwise its below 100%. This same value `daysp` is seen in the last two columns of the next set of difficulties. Here, the key is next set of difficulties.

How Long does it Take to Mine a Block?

```
ch0717.py
diff = 20000
hrate = 10**9
secs = 60.0 * 60
time = diff * 2**32 / hrate / secs
print time
```

```
Output
23.8608333333
```


The formula to calculate the hash rate is as follows:

$$\text{hashrate} = \text{difficulty} * 2^{32} / \text{time}$$

The time is in 600 seconds.

$$\text{time} = \text{difficulty} * 2^{32} / \text{hashrate}$$

The hash rate of 1 GHz and a difficulty of 20,000, (Wiki values) are divided by 3600 to get a value in hours and not seconds. If the mining rig is running this slow, it would take the above difficulty approximately 24 hours to mine a block. Nevertheless, any hash value can meet this condition. Mining a block is a random event. Thus, it comes as no surprise that individual miners are extinct in the world of miners as the hardware is worth tens of millions in hardware.

Another technicality here about the target is that the sha hash value is compared with the number of 0's in the target. The number of 0's in front of a number decides the value of the number, more the leading 0's smaller is the number.

Inserting Millions of Transactions in a SQL Table

First create a simple table in psql.

```
create table trans (fno integer , blockno integer , tranno integer , thash varchar(64));
```

ch0718.py

```
from cfuncs2 import *
```

```
import pycpg2
```

```
def doutput(f):
```

```
    nooutputs = rvarint(f)
```

```
    outputarr = []
```

```
    for i in range ( 0 , nooutputs):
```

```
        bitcoinvalue = r8bytes(f)
```

```
        outputscriptlen = rvarint(f)
```

```
        scriptpubkey = rbytes(f , outputscriptlen).encode('hex')
```

```
        output = (bitcoinvalue , outputscriptlen , scriptpubkey)
```

```
        outputarr.append(output)
```

```
    return outputarr
```

```
def dinput(f):
```

```
    noinputs = rvarint(f)
```

```
    inputarr = []
```

```
    for i in range ( 0 , noinputs):
```

```
        prevtrhash = rhash(f)
```

```
        outputindex = rint(f)
```

```
        inputscripplen = rvarint(f)
```

```
        sigpubkey = rbytes(f , inputscripplen).encode('hex')
```

```
        seqno = rint(f)
```

```
        input = (prevtrhash[:-1], outputindex, inputscripplen, sigpubkey, seqno)
```

```
        inputarr.append(input)
```

```
    return inputarr
```

```
blockno = 0
```

```
tranno = 0
```

```
conn = pycpg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
```

```
cur = conn.cursor()
for fno in range(0, 1000 ):
    fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat" % fno
    print "%s tranno %d blockno %d " % (fname, tranno , blockno)
    try:
        f = open(fname , "rb")
    except:
        break
    while (True):
        blockno = blockno + 1
        try:
            magic = rint(f)
            if magic == 0:
                continue
            except:
                break
            size = rint(f)
            header = f.read(80)
            notran = rvarint(f)
            for tno in range ( 0 , notran):
                tranno = tranno + 1
                startfp = f.tell()
                tver = rint(f)
                inputa = dinput(f )
                outputa = doutput(f)
                locktime = rint(f)
                endfp = f.tell()
                f.seek(- (endfp - startfp) , 1)
                trand = f.read(endfp - startfp)
                thash = chash(trand)
                thash = thash[:-1].encode('hex')
                s1 = "insert into trans values (%d , %d, %d , '%s') " % (fno , blockno , tranno , thash)
                cur.execute(s1)
            conn.commit()
```

This program takes an entire night to execute. The total number of transactions inserted on the table trans is a whopping 142,672,269, which is over 146 million, and the mammoth task is performed in less than 8 hrs.

Every transaction found in the Bitcoin blocks is a row in the table. The transaction hash value and the file number of the block file, a virtual block and transaction number are also added. At the end of the program, there are about 140 million transactions/ rows in the table

Then a simple sql query is executed which discloses a count of duplicate transactions or transactions having the same hash value.

```
postgres=# select thash , count(*) from trans group by thash having count(*) > 1 limit 2;
 thash | count
-----+-----
000873e5f13e6327ae123780cbff53befd4ec949ca885514e672a89850b0549b | 2
0024c686611fe275651cf611803115c8d74d53083522b0fc61f49c9fe4df0bd8 | 2
```

This is standard SQL select command for ferreting out duplicates in columns. The group by works with the same column value, in this case, thash. The having works like a where clause, but only on the results returned by a 'group by' clause. Once again your mileage will be different.

This sql statement retrieves rows having duplicate values. To our utter surprise, there are about a 100 duplicate transaction hashes.

Something wrong here as a SHA hash guarantees that you will get no collisions.

```
select * from trans where thash
='000873e5f13e6327ae123780cbff53befd4ec949ca885514e672a89850b0549b' limit 2;
fno | blockno | tranno | thash
555 | 418588 | 138593075 |
000873e5f13e6327ae123780cbff53befd4ec949ca885514e672a89850b0549b
555 | 418589 | 138595479 |
000873e5f13e6327ae123780cbff53befd4ec949ca885514e672a89850b0549b
```

To reconfirm, we execute the above query. The output confirms the fact that transaction hash 000873 is present twice in two different blocks but in the same blk file starting with 555. Two transactions having the same hash is called transaction malleability.

We have written all the code in this book ourselves and executed all the programs as well. There is no reason for a program that adds over 150 million transactions in a database should fail. But fail it did.

When you run this program, you will get an exception thrown on block number 976. The date of the block creation was Thursday the 24th of August at 07:27:37. But we expected it. Everything is as per plan. If this code worked, then all hell will break loose.

Transaction Inputs and Outputs for any Transaction having a Hash Value Starting with 000873

```
ch0719.py
from cfuncs2 import *
import pybitcointools
def doutput(f ):
    nooutputs = rvarint(f)
    outputarr = []
    for i in range ( 0 , nooutputs):
        bitcoinvalue = r8bytes(f)
        outputscriptlen = rvarint(f)
        scriptpubkey = rbytes(f , outputscriptlen).encode('hex')
        output = (bitcoinvalue , outputscriptlen , scriptpubkey)
        outputarr.append(output)
    return outputarr
def dinput(f ):
    noinputs = rvarint(f)
    inputarr = []
    for i in range ( 0 , noinputs):
        prevtrhash = rhash(f)
        outputindex = rint(f)
        inputscripplen = rvarint(f)
```

```
    sigpubkey = rbytes(f , inputscriptlen).encode('hex')
    seqno = rint(f)
    input = (prevtranhsh[::-1], outputindex, inputscriptlen, sigpubkey, seqno)
    inputarr.append(input)
    return inputarr
blockno = 0
tranno = 0
for fno in range(555 , 556 ):
    fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat" % fno
    #print "%s tranno %d blockno %d " % (fname, tranno , blockno)
    try:
        f = open(fname , "rb")
    except:
        break
    while (True):
        blockno = blockno + 1
        try:
            startb = f.tell()
            magic = rint(f)
            if magic == 0:
                continue
            except:
                break
            size = rint(f)
            header = f.read(80)
            bhash = chash(header)
            bhash = bhash[::-1].encode('hex')
            notran = rvarint(f)
            for tno in range ( 0 , notran):
                tranno = tranno + 1
                startfp = f.tell()
                tver = rint(f)
                inputa = dinput(f )
                outputa = doutput(f)
                locktime = rint(f)
                endfp = f.tell()
                f.seek(- (endfp - startfp) , 1)
                trand = f.read(endfp - startfp)
                thash = chash(trand)
                thash = thash[::-1].encode('hex')
                if thash == '000873e5f13e6327ae123780cbff53befd4ec949ca885514e672a89850b0549b':
                    print "Block Hash is %s" % bhash
                    print "Transaction Hash is %s" % thash
                    print "fno %d blockno %d tranno %d start tno % d transaction %d block %d" % (fno , blockno , tranno , tno ,
                    startfp , startb)
                    print "No of Inputs %d" % len(inputa)
                    cnt = 0
```

```

for input in inputa:
    print "Input Number %d" % cnt
    print "\tPrevtransatcion Hash %s:%d" % (input[0].encode('hex') , input[1])
    print "\tSigPublicKey %s:%d" % (input[3] , input[2])
    print "\tSequence Number %d" % input[4]
    cnt = cnt + 1
print "Number of Outputs %d" % len(outputa)
cnt = 0
for output in outputa:
    print "Output Number %d" % cnt
    print "bitcoin Value %s" % output[0]
    print "Script Public Key %s:%d" % (output[2] , output[1])
    print
    print

```

Output

Block Hash is

000000000000000004b33fcc99679846a5baf29896d2f9d4e52ec3fe617b7eaa

Transaction Hash is

000873e5f13e6327ae123780cbff53befd4ec949ca885514e672a89850b0549b

fno 555 blockno 136 tranno 172398 start tno 121 transaction 104870818 block 104754391

No of Inputs 1

Input Number 0

Prevtransatcion Hash

fc2c5823d20c71595bef1d6537a084e08445abd9e7ce4c0d9125143d876de8e6:1

SigPublicKey

00483045022100de388be2a4ecb60fbaabcb7b9f5f5d4b1f70a14e19216f097d675ea714
ce851602204d106d8c38665be92c16442231c94ea4e547b291fdfe8419a16c5368f277c1
2e01483045022100ec646f48c8f660139bc47c9db8d20c860f1ed53b49a28784eeec1f8bf
03cf0b202200e9baa6105b8d0306f62fd36da8a9a319f4c88feff268dda68d7da6c7604f3
860147522102a6b211f07f9ec2a1b8ee51ac6459b878ded8c2b9f3e1b599e3e794ec58a4
332d21036a5ca3ecb480fb5a930c4875a32d5eeb585effec73e68753f0de7962f474b97b
52ae:219

Sequence Number 4294967295

Number of Outputs 2

Output Number 0

bitcoin Value 1000000

Script Public Key a91462e037114eaa8ed465d2f4c89041b60025619f2187:23

Output Number 0

bitcoin Value 993142311

Script Public Key a914f211b44778226ea4f01c4634c5728d8becdb72f487:23

Block Hash is

0000000000000000047ffadd5468b3dd74b87b761a4c77724b57e259b5d51ded

Transaction Hash is

000873e5f13e6327ae123780cbff53befd4ec949ca885514e672a89850b0549b

fno 555 blockno 137 tranno 174802 start tno 118 transaction 105780756 block 105742467

```
No of Inputs 1
Input Number 0
  Prevtransacton Hash
fc2c5823d20c71595bef1d6537a084e08445abd9e7ce4c0d9125143d876de8e6:1
  SigPublicKey
00483045022100de388be2a4ecb60fbaabcb7b9f5f5d4b1f70a14e19216f097d675ea714
ce851602204d106d8c38665be92c16442231c94ea4e547b291fdfe8419a16c5368f277c1
2e01483045022100ec646f48c8f660139bc47c9db8d20c860f1ed53b49a28784eeec1f8bf
03cf0b202200e9baa6105b8d0306f62fd36da8a9a319f4c88feff268dda68d7da6c7604f3
860147522102a6b211f07f9ec2a1b8ee51ac6459b878ded8c2b9f3e1b599e3e794ec58a4
332d21036a5ca3ecb480fb5a930c4875a32d5eeb585effec73e68753f0de7962f474b97b
52ae:219
  Sequence Number 4294967295
Number of Outputs 2
Output Number 0
  bitcoin Value 1000000
  Script Public Key a91462e037114eaa8ed465d2f4c89041b60025619f2187:23
Output Number 0
  bitcoin Value 993142311
  Script Public Key a914f211b44778226ea4f01c4634c5728d8becdb72f487:23
```

Here, there are orphan blocks and not transaction malleability

In this program, using sql statements, the transaction inputs and outputs for any transaction having a hash value starting with 000873 is displayed.

First, it is verified that the two transactions have the same inputs and outputs, bit by bit. The possibility comes in only when one of the transactions in file blk00555.dat is in an orphan block. We know the file name and the block number because we had cross checked earlier, it was more to avoid wasting any more time. The program may not show the same result to you as your files and blocks are different. As explained earlier, orphan blocks are added to the Bitcoin block files.

Let's look at output in blockchain.info. Please enter the following transaction in blockchain.info.

```
000873e5f13e6327ae123780cbff53befd4ec949ca885514e672a89850b0549b
```

The web page shows two blocks with the same transaction and both the blocks numbers are the same, 418023. When we right click on each of these numbers and open the block details in a separate browser tab, it reveals that the block numbers are the same but the block hashes are different. They, however, match the block hashes our program displays. The block hash ending with 7eaa is shown as the orphaned block.

It does get confusing initially when we see the same transaction hash twice, we think that the Sha256 hash is broken. Also, we see orphan blocks been given block numbers that clash with existing block numbers.

It is best to conclude saying, do what the bitcoin world wants you to do, use pruning to make sure that you are not saddled with 200 GB of blocks that are of no use to anyone, whatsoever. We will cover pruning later.

CHAPTER 08

Transactions - Inputs and Outputs

So far, we have learnt that a Bitcoin block starts with a 4-byte magic number, followed by another 4 bytes for the block size and then block data. The block data has an 80-byte block header, then a couple of bytes giving a count value, i.e. number of transactions in the block and finally the transactions. Every transaction has an input and outputs.

In this chapter, let us further elaborate on the concepts of transaction inputs and transaction outputs, learnt in chapter 2. All said and done, these transactions are the most important part of the blockchain technology and not the block header.

Our focus in this chapter is understanding the structure of a Bitcoin Transaction. Therefore, let's start explaining these transactions from the very beginning. The Bitcoin transactions are stored on a computer in a file on disk. Having one large file would be unwieldy and therefore multiple .dat files, 128 MB large, are created. However, there is no rule that states the files should be only 128 Megabytes large. Similarly, there is no rule on the number of blocks either. But still files are only 128 MB large as this number is hardcoded in the Bitcoin source code.

Mr. Satoshi could have chosen the option of storing Bitcoin transaction data in a database in place of multiple .dat files. Further, there is no explanation on why the Bitcoin community does not use an open source sql or NoSQL data base to store these massive volumes of Bitcoin data. Only the Bitcoin source has the answers to these questions. The Bitcoin community chooses embedded databases like leveldb to store some Bitcoin data. The data in these embedded databases are more important than the transaction data.

The size of the .dat file on disk is already over 200 GB large and growing.

Let's make a very rough calculation. A Bitcoin block is created every 10 minutes, this means 6 blocks an hour. In 24 hours or 1 day, there will be $6 * 24$ or 144 blocks. So, in a month, assuming 30 days, there will be $144 * 30$ or 4320 blocks. And in a year of 365 days the count will be $4320 * 12 = 51840$ blocks a year.

Each block is a maximum of 1MB in size, therefore it is 51840 MB a year in storage. Dividing by 1000 gives 51GB, the amount of storage being added every year. At the start of the year 2018, the blocks would take up over 200 GB of storage. This is on the assumption that every block is full and would use the entire 1MB of storage. The block size can now be larger, up to 8MB for some Bitcoin blockchains. We have 3 different forks and still counting. Nevertheless, today there is a waiting list or a queue to add transactions to the block and people pay more to a miner to see their transactions come through.

This 1MB block limit is the biggest controversy ever that has divided the Bitcoin community. Despite all debates and noise, the Bitcoin community has failed to agree on one best block size. People have written books on this subject and it seems there is and will be no consensus. Everyone has a point of view and they are all very right in their thinking.

The first transaction in a block is created by the miner. It basically contains his fee for work done or finding the right hash. This fee started off at 50 Bitcoins, and then reduced to 25 Bitcoins. At the time of writing this chapter in the book, the miner fee is down to 12.5 Bitcoins. The world has remained oblivious to this change.

The multiple inputs in a transaction specify the addresses the Bitcoins are coming from and the multiple outputs state the addresses the Bitcoins are going to.

There are many inputs in a transaction, thus a transaction has many sources. These individual inputs must refer to an output in the past. These inputs are then transferred to either one output or Bitcoin address or multiple addresses. At the end of the day, a transaction is moving Bitcoins from one address/es to another address/es.

The inputs specify the address of a transaction and an index to an Output which has in the past received bitcoins. In other words, the input refers to a previous Output that has been unspent.

The input Bitcoins will always be greater than the output's Bitcoins. The difference, also called change will be paid to the miner for his efforts. As per the Bitcoin guidelines, the inputs cannot be less than the outputs as no one can give more than what they have. If they are equal, then miners do not get paid and the transactions will get mined in the 25th century only.

There is another way to pay the miner. As the Bitcoin reward drops over time, the change received by the miner grows as a percentage of the miner's earnings.

A Bitcoin transaction is like a ledger which records the source from where money is coming and the destination, where it is going. Therefore, a Bitcoin cannot be bought or sold but can only have its ownership transferred between Bitcoin addresses.

An Output contains a Bitcoin value and information of the current Bitcoin's owner. However, there is one more aspect to the Outputs. Ditto for inputs, they contain more than just a link to an output. This will be explained in course of time.

It took us a long time to understand these fundamentals. We assume that you too fall in the same bracket. That's the reason, we keep explaining inputs and outputs in different ways.

Before we start, let's write a series of functions that displays an entire transaction, given its hash value.

Displaying a Transaction Details Given a Bblock Number and Hash

```
cfuncs1.py
from cfuncs import *
def doutput(f):
    nooutputs = rvarint(f)
    outputarr = []
    for i in range ( 0 , nooutputs):
        bitcoinvalue = r8bytes(f)
        outputscriptlen = rvarint(f)
        scriptpubkey = rbytes(f , outputscriptlen).encode('hex')
        output = (bitcoinvalue , outputscriptlen , scriptpubkey)
        outputarr.append(output)
    return outputarr

def dinput(f):
    noinputs = rvarint(f)
    inputarr = []
    for i in range ( 0 , noinputs):
        prevtrhash = rhash(f)
        outputindex = rint(f)
        inputscripflen = rvarint(f)
        sigpubkey = rbytes(f , inputscripflen).encode('hex')
```



```

    seqno = rint(f)
    input = (prevtrhash[:::-1], outputindex, inputscripflen, sigpubkey, seqno)
    inputarr.append(input)
    return inputarr

def tran(phash , start):
    for fno in range(start , -1 , -1):
        fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat" % fno
        #print fname
        try:
            f = open(fname , "rb")
        except:
            break
        while (True):
            try:
                magic = rint(f)
                if magic == 0:
                    continue
            except:
                break
            size = rint(f)
            header = f.read(80)
            notran = rvarint(f)
            for tno in range ( 0 , notran):
                startfp = f.tell()
                tver = rint(f)
                inputa = dinput(f )
                outputa = doutput(f)
                locktime = rint(f)
                endfp = f.tell()
                f.seek(- (endfp - startfp) , 1)
                trand = f.read(endfp - startfp)
                thash = chash(trand)
                thash = thash[:::-1].encode('hex')
                if thash == phash:
                    return (fno , inputa , outputa , locktime)
            return (-1 , [], [], 0)

```

The function tran does all the heavy weight lifting work once it is given a hash value and the block number of a transaction.

The initial code that scans every file and every block in the dat file remains the same. As always, the magic number and the size of the data is read but it can be ignored. The intention is to read every byte of the block data. The block header fields are also read and then ignored.

What's most important is the number of transactions in the block. The function rvarint is used for this purpose. The number of bytes storing this value keeps changing. It is saved in variable notran. The for loop reads the data of all the sequentially ordered transactions in the block.

The transaction data in the block is the last entity stored in the block. There is nothing stored after it. The size of each transaction remains a mystery as nowhere in the transaction data is a field where the size is mentioned. Each transaction will have a different size. This complicates the writing of code, as we have no direct way of finding out the size of the transaction. The size of every transaction keeps changing.

Let's start with the structure of one transaction. Every transaction starts with a version field that is ignored for now. Then is a series of inputs followed by a series of outputs and it all ends with a field called locktime. Earlier the functions, `dinput` and `doutput`, read and displayed the inputs and outputs respectively. Now we use a modified version of these functions. They only read the inputs and outputs, nothing is displayed. The functions return a list of inputs and outputs. The field locktime marks the end of a transaction.

Before the reading of every transaction record, the file pointer is stored in a variable called `startfp`. Similarly, after reading the last field called locktime, the file pointer is stored in variable `endfp`. The difference `endfp - startfp` determines the size of the current transaction. The seek function with -ve sign jumps backwards to the start of the transaction data. Once the size is figured, the read function is used to store the entire transaction data in variable `trand`.

Thereafter, the function `chash` calculates the transaction hash. This transaction hash must be computed for every transaction, it is not stored as part of the transaction data. As observed earlier, sites like `blockchain.info` prefer the reverse of a hash, so we have no choice but to reverse the hash by using the slice operator `[::-1]`.

The inputs and outputs in a transaction start with the number that tells us how many inputs or outputs are following, the functions `dinput` and `doutput` start with an empty array or a list. All individual fields of an output or input are collected into a tuple and the append function adds to this list. We would like to remind you once again, that the inputs and outputs also start with a variable number of bytes giving a count on the inputs or outputs respectively. The list in the input category has 5 values and there are 3 values for the output.

When a transaction hash matches the anticipated hash value, we return the following values, the physical file number, the lists of inputs and outputs and finally the locktime field value. If there is no match, we return -1, an empty list for the inputs and outputs and 0 for the locktime field.

In the function `tran`, earlier, we started with a value 0 and then moved up to a large value like 10000. Here, the loop starts with the value in the parameter. The loop moves backwards and not forwards as the last parameter to the range function is -1. It ends when the `fno` variable reaches a value of 0, at file name `blk00000.dat` and not `blk0000-1.dat`. So, if the parameter `start` has a value of 100, file `blk00100.dat` will be the first file to be read and the loop ends with file `blk00000.dat`. In this manner, every transaction in this file is read.

There is some method behind our madness. The objective here is to not only display one input transaction but also display the output of the transaction that the input refers to.

The inputs point to a transaction that supplies the outputs. The output is also displayed with the inputs for every transaction. It brings in more clarity.

If there is a transaction which is mined say in `block00500.dat`, chances are that the transaction hashes that are referred to in the input section will come from a transaction stored say in `blk00490.dat`. It may not come from transactions stored in `blk00000.dat`. The assumption is that transactions will get their Bitcoins from outputs in later transactions than from the earliest transactions. Therefore, the search is backwards and not from the start.

Displaying a Transaction with the Outputs Referred by the Transaction Inputs

```
ch0801.py
from cfuncs1 import *
thash = 'cca7507897abc89628f450e8b1e0c6fca4ec3f7b34cccf55f3f531c659ff4d79'
(fno , inputa , outputa , locktime ) = tran(thash , 0)
```

```

print "Transaction Hash is %s" % thash
for i in range(0, len(inputa)):
    print "\tInput Number %d" % i
    print "\t\tTransaction Hash is %s" % inputa[i][0].encode('hex')
    print "\t\tOutput Index in transaction %x" % inputa[i][1]

    (fno1, inputa1, outputa1, locktime1) = tran(inputa[i][0].encode('hex'), 0)
    for j in range(0, len(outputa1)):
        if j == inputa[i][1]:
            print "\tOutput Number %d" % j
            print "\t\tValue of the Bitcoin %d:%.02f" % (outputa1[j][0], outputa1[j][0] * 1.0/100000000.00)
            print "\t\tLength of Output Script %d" % outputa1[j][1]
            print "\t\tOutput Public Key Script %s" % outputa1[j][2]

            print "\t\tInput Script Length ScriptSignature %d" % inputa[i][2]
            print "\t\tSignature + Public Key %s" % inputa[i][3]
            print "\t\tSequence Number %x" % inputa[i][4]
for i in range(0, len(outputa)):
    print "\tOutput Number %d" % i
    print "\t\tValue of the Bitcoin %d:%.02f" % (outputa[i][0], outputa[i][0] * 1.0/100000000.00)
    print "\t\tLength of Output Script %d" % outputa[i][1]
    print "\t\tOutput Public Key Script %s" % outputa[i][2]

```

Output

```

Transaction Hash is
cca7507897abc89628f450e8b1e0c6fca4ec3f7b34cccf55f3f531c659ff4d79
    Input Number 0
    Transaction Hash is
a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d
    Output Index in transaction 0
    Output Number 0
    Value of the Bitcoin 1000000000000:10000.00
    Length of Output Script 25
    Output Public Key Script
76a91446af3fb481837fadbb421727f9959c2d32a3682988ac
    Input Script Length ScriptSignature 139
    Signature + Public Key
4830450221009908144ca6539e09512b9295c8a27050d478fbb96f8addbc3d075544dc41
328702201aa528be2b907d316d2da068dd9eb1e23243d97e444d59290d2fddf25269ee0
e0141042e930f39ba62c6534ee98ed20ca98959d34aa9e057cda01cfd422c6bab3667b76
426529382c23f42b9b08d7832d4fee1d6b437a8526e59667ce9c4e9dcebcabb
    Sequence Number ffffffff

    Output Number 0
    Value of the Bitcoin 577700000000:5777.00
    Length of Output Script 25
    Output Public Key Script
76a914df1bd49a6c9e34dfa8631f2c54cf39986027501b88ac
    Output Number 1

```

```
Value of the Bitcoin 422300000000:4223.00
Length of Output Script 67
Output Public Key Script
4104cd5e9726e6afeae357b1806be25a4c3d3811775835d235417ea746b7db9eeab33cf
01674b944c64561ce3388fa1abd0fa88b06c44ce81e2234aa70fe578d455dac
```

The transaction details are displayed of a transaction hash starting with cca7. This transaction has one input and two outputs. The function tran is called with the transaction hash and the block number, 0.

The output displayed is a list of tuples returned from the tran function. The two for loops iterate on tuples, inputa and outputa.

In the for loops, the index variable i accesses each input and output. The same array notation [] with the index variable i is used to access each row in the list. The array notation [] is also used to access each member within the tuple and the tuple is displayed.

Displaying a transaction in this manner showcases only a part of the big picture. The two outputs in the transaction give information on everything we need to know, like the addresses of the new owners of the Bitcoins.

The inputs however project very limited information. It discloses the fact that the Bitcoins are coming from an output, 0 of a transaction hash that starts with a105. It gives no data on the transaction supplying the Bitcoins.

Let's modify things to make sense of it. First, the transaction hash and the output index number is displayed. Next, the tran function is called once again but with the transaction hash found in the input. This hash starts with 1075 and is stored in a list element, inputa[i][0]. Since the transaction hash is the first member of the tuple, the slice operator has 0 in the list.

A transaction may have zillion outputs. We only display the output referred by the current input's output index field. This transaction is in block 57043 which is present in the file blk00000.dat. Therefore, the file number is 0 in both function calls to the tran function.

Let's modify this further to make more sense of the Inputs.

Verifying that the Input Public Key of a Transaction Matches the RipeMD Hash of the Output

```
ch0802.py
from cfuncs1 import *
import pybitcointools
thash = 'cca7507897abc89628f450e8b1e0c6fca4ec3f7b34cccf55f3f531c659ff4d79'
print thash
(fno , inputa , outputa , locktime ) = tran(thash , 0)
sigpub = inputa[0][3].decode('hex')
print len(sigpub)
print sigpub.encode('hex')
siglen = ord(sigpub[0])
print "Sig Len %d" % siglen
sig = sigpub[1:siglen+1]
sige = sig.encode('hex')
print "Sig %s" % sige
publen = ord(sigpub[siglen + 1])
print "Pub len %d" % publen
puba = sigpub[siglen+2: siglen+1+ publen+1]
```

```

pub = puba.encode('hex')
print "Pub %s" % pub
hash = pybitcointools.hash160(puba)
print "RipeMD %s" % (hash)
baddr = pybitcointools.hex_to_b58check(hash)
print "Bitcoin address %s" % baddr
print
(fno1, inputa1, outputa1, locktime1) = tran(inputa[0][0].encode('hex'), 0)
scripub = outputa1[inputa[0][1]][2]
print "inputa[0][1] ", inputa[0][1]
print "outputa1[inputa[0][1]] ", outputa1[inputa[0][1]]
print "outputa1[inputa[0][1]][2] ", outputa1[inputa[0][1]][2]
print "OScript %s" % scripub
print "OScript Length %d" % len(scripub)
scripubd = scripub.decode('hex')
scriptstr = "Plen %d " % ord(scripubd[2])
opcode = ord(scripubd[0])
print "First Opcode %x" % opcode
ripemd = scripub[6:46]
baddri = pybitcointools.hex_to_b58check(ripemd)
if opcode == 0x76:
    print "OP_DUP ",
    opcode = ord(scripubd[1])
if opcode == 0xa9:
    print "OP_HASH160 ",
    print "%x" % ord(scripubd[2]),
    print "%s " % ripemd,
    opcode = ord(scripubd[23])
if opcode == 0x88:
    print "OP_EQUALVERIFY ",
    opcode = ord(scripubd[24])
if opcode == 0xac:
    print "OP_CHECKSIG "
print "Bitcoin Address %s" % baddri
if baddri == baddr:
    print "We have a match"
else:
    print "vijay Mukhi is a embecile"

```

Output

```

cca7507897abc89628f450e8b1e0c6fca4ec3f7b34cccf55f3f531c659ff4d79
139
4830450221009908144ca6539e09512b9295c8a27050d478fbb96f8addbc3d075544dc41
328702201aa528be2b907d316d2da068dd9eb1e23243d97e444d59290d2fddf25269ee0
e0141042e930f39ba62c6534ee98ed20ca98959d34aa9e057cda01cfd422c6bab3667b76
426529382c23f42b9b08d7832d4fee1d6b437a8526e59667ce9c4e9dcebcabb
Sig Len 72

```

```
Sig
30450221009908144ca6539e09512b9295c8a27050d478fbb96f8addbc3d075544dc4132
8702201aa528be2b907d316d2da068dd9eb1e23243d97e444d59290d2fddf25269ee0e01
Pub len 65
Pub
042e930f39ba62c6534ee98ed20ca98959d34aa9e057cda01cfd422c6bab3667b7642652
9382c23f42b9b08d7832d4fee1d6b437a8526e59667ce9c4e9dcebcabb
RipeMD 46af3fb481837fadbb421727f9959c2d32a36829
Bitcoin address 17SkEw2md5avVNyYgj6RiXuQKNwkXaxFyQ

inputa[0][1]] 0
outputa1[inputa[0][1]] (1000000000000, 25,
'76a91446af3fb481837fadbb421727f9959c2d32a3682988ac')
outputa1[inputa[0][1]][2] 76a91446af3fb481837fadbb421727f9959c2d32a3682988ac
OScript 76a91446af3fb481837fadbb421727f9959c2d32a3682988ac
Oscript Length 50
First Opcode 76
OP_DUP OP_HASH160 14 46af3fb481837fadbb421727f9959c2d32a36829
OP_EQUALVERIFY OP_CHECKSIG
Bitcoin Address 17SkEw2md5avVNyYgj6RiXuQKNwkXaxFyQ
We have a match
```

The explanation given for this program is much longer than the program itself. Maybe the longest explanation so far.

Let's first examine an input of a transaction. Outputs are much simpler to comprehend. The first task is to display just one transaction's input without displaying the block header, the outputs and any the other transaction details etc. The transaction hash in the earlier program is used again:

```
cca7507897abc89628f450e8b1e0c6fca4ec3f7b34cccf55f3f531c659ff4d79.
```

This hash value has a special place as it is the first transaction with one input and two outputs. It is the smallest transaction and we are comfortable with it.

An input gets money on the table. An input represents incoming bitcoin money that will eventually be transferred to someone. It must reference a prior output. A fundamental reason is that you cannot transfer money to anyone if you have not received money earlier from someone. Money is synonymous to Bitcoins. The other way out is to be a miner. There are only two known ways today, to be a proud owner of Bitcoins, either by being a miner or by purchasing them. We have not met a Bitcoin miner so far as there are very few of them in existence today, mostly Chinese. To be a miner and successfully crack the hash requires machines that cost in the millions of dollars.

For us, the first interesting member in the inputs is the transaction hash. It contains specific information of the sender of Bitcoins, who brings in the Bitcoins whose ownership needs to be transferred. In our case this hash is

```
a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d.
```

In the Bitcoin world, the original hash is always reversed. However, we choose to display the hash as a little endian or a big-endian number, depending upon our mood. A transaction is known by its hash value, though it is not a field in the Bitcoin transaction data structure. That's because, it is a computed value. It uniquely represents a transaction. Later, we will talk about transaction malleability also.

This transaction hash starting with 'a1075' may have multiple outputs. Therefore, the next integer in the inputs is the

index or offset in the output of the above transaction. This is the only way to identify and isolate the output amongst the others. This output is the source of the Bitcoins for this input.

Then comes the length of the input script. This field normally has a value of 139 but can vary. The field is called ScriptSig in the Bitcoin literature, short form of SignatureScript. For the record, the script length is not part of the input structure as it is variable in length and contributes nothing new. The 139 bytes are saved in the variable, sigpub. If the ScriptSig field had a fixed size then the length field becomes redundant.

These 139 bytes contain the signature of the current transaction and more. A signature is very similar to a hash plus more. Like a hash it represents some data, but unlike a hash it verifies an owner.

In a previous chapter, we generated a random number called private key. The same concept is applied here. Our random lucky number for the private key is 420. Then using a function from the pybitcointools library, a public key is generated from this private key. It is based on Elliptic Curve Cryptography.

This key is called a public key as it is made public to the world, unlike a private key. The public key in this form of cryptography is generated from the private key only. Given a public key, there is no way to arrive at the private key, as this transformation is only one way.

In Elliptic Curve Cryptography or ECC, some data called a message (in this case our Bitcoin transaction data) is passed through some code that uses a private key only and it generates an output, which is a signature. This signature is unique to the message or Bitcoin transaction data. Any change, even one bit to the message and the signature changes dramatically. Unlike a hash, this signature depends upon the private key used and the data it has signed.

The same message and signature when decrypted using the public key, confirms the authenticity of the message being signed by the private key.

It must be noted that three values are given to this code, the original message, the public key and the original signature. The private key is not included in this equation as only the owner has it. The owner will never make this private key public. Once again, the public key is only derived from the private key.

In other words, the transaction is signed by a certain private key only and no one has any access to it. A signature verifies if a certain message has been signed by a private key, but without holding one. We will understand signatures only when we create our own signature and verify that our private key has signed this data.

A signature announces to the world who the rightful owners of the Bitcoin are as they own the private key. It is thus concluded that the private key determines the ownership of a bitcoin, not the Bitcoin address itself. If the private key is hacked, then the Bitcoins get seized automatically.

A public key comes from a private key. Then the SHA hash value is calculated and thereafter comes the RipeMD hash. Finally, there is the base58 encoding with a checksum. After repeating this ad nauseous, we hope you are finally getting a hold on this information.

From the public key comes the Bitcoin address. Every input has a field called ScriptSig, which contains the signature, followed by the public key.

A two-stage process is followed. First the signatures must match. Then the public key in the input or ScriptSig must resolve to the Bitcoin address in the Output. This verifies and discloses the fact that a private key has signed the transaction. Plus, it reassures the world that the public key created this Bitcoin address. This whole process is foolproof as the conversion of a private key to public key to Bitcoin address is only one way.

It is something very fundamental. Only the owner of a bitcoin can transfer the ownership to someone else. With this process in place, one can verify the true owner of Bitcoins.

Bitcoins are present in the value field of the output of a transaction when the ownership is transferred from one entity to another. Let's say A transferred ownership of Bitcoins to B. Now, the output field of the transaction will have the details

of the Bitcoin value and the Bitcoin address or equivalent of, B. Also, A must prove ownership of the bitcoins when they are transferred. The signatures play a crucial role here.

A step by step calculation of the signature using program code will help understand this concept better. That will be in another chapter.

As mentioned earlier, the ScriptSig field contains two separate values. First is the signature followed by the public key. The signature has a variable length because the signature value keeps changing when re-generated. We lost Bitcoins because our signatures did not change. So, the signature value will change if the signature is calculated twice even when the same message and same private key are used.

The third line of the output has information on the signature. In our output, the first byte of the signature is the length of the signature, 48 or 72 in decimal. It is followed by the signature bytes, 72 of them. The next 144 characters make the signature. It ends with the bytes 0e01. Then comes the length of the public key, 0x41 or 65 bytes or 130 characters. The public key here, starts with the bytes 042e and end with cabb.

We must clarify a few things here. By our logic, the signature and public key have a byte to store the length. This limits the length to 255 bytes. Nevertheless, the length is stored in a more complex manner to accommodate larger values. More on it later.

Having seen the output, let's look at the code, one line at a time and understand it better. There have been special cases where there is no public key.

The variable sigpub stores the ScriptSig value as a decoded string. The inputs list stores the ScriptSig field as an encoded string. Other than using it for display purposes, there is not much use of it here. This decoded variable sigpub now has numbers ranging from 0 to 255 but it is stored as a string variable. The ord function returns the decimal values in this string.

The first byte in the sigpub variable has a value of 72. This value is stored in the variable siglen. It indicates that the next 72 bytes starting from the characters 3045 is the signature of the transaction. Using the signature to verify the owner of the Bitcoin address is left for the subsequent chapter.

The python slice operator ' : ' extracts the next 72 bytes. The operator is given 1 for the 0th byte and ends with 73, the slice operator stops at one less value. For this purpose, we add one to siglen. The variable sig stores the signature. This string is then decoded. The variable sig has the decoded signature and sig, the encoded signature.

The byte following the signature is the length of the public key. In the program, this byte i.e. the 73rd one is accessed by adding 1 to the variable siglen. The variable publen stores this length. The public key is stored uncompressed; the length is 65 bytes.

Now comes the tricky part of extracting the public key. We start not from variable siglen but 2 bytes ahead. The reason being that both the lengths of the public key and the length of the signature which take up one byte each are to be accounted for.

The slice operator looks a little complex. To extract the public key, the start index is placed at the length byte + 2, the actual public key starts after the length byte. Also, it cannot be assumed that the public key continues till the end of the string. Thus, the ending point is the length of the signature and public key + 2, which are the length bytes. The variable puba stores the decoded public key.

The encoded public key is saved in the pub variable. The variable pub shows the same value as the original ScriptSig field, when displayed. The function hash160 from the pybitcointools module returns a SHA hash value of the key. This is internally followed by the computation of the RipeMD hash. The RIPE MD hash is displayed, and as you see, it is nowhere close to the final Bitcoin address. This hash has no meaning to anyone. The decoded variable, puba, is generally used everywhere and not the encoded one.

Next comes the Bitcoin address, which is a base58 encoded string and a checksum tacked at the end. To accomplish this, the `pybitcointools hex_to_base58check` function is used. The return value, as in Bitcoin address is saved in a variable `baddr` and then displayed. The Bitcoin address starts with `17Sk`.

All that the input offers so far, is data, a signature and a public key. But, some private key signed the transaction data to give this signature. So, let's decipher this private key.

The 65-byte public key stored in the inputs gives a Bitcoin address. The transaction hash is a part of the input. It is the first column in the list, `inputa`. The same function `tran` is now used to obtain the outputs from the transaction hash. Retrieving this output is a little tricky. The variable `inputa[0][1]` refers to the output index in the list of outputs. In our case this value is 0.

The variable `outputa1[inputa[0][1]]` has the entire output record with 3 fields. It must be noted the input points to these outputs. Our interest is in the second member of this tuple (it is extracted using the expression `outputa1[inputa[0][1]][2]`). It is the script public key member. The variable `scrpub` contains the `ScripPubKey` (as called by the Bitcoin wiki). The script length of most outputs are 25 or 50 depending upon data being decoded or encoded.

It is not a 'chicken and egg' story because the output referenced by the transaction hash in the input appears earlier in time. The reason being that you cannot pass the ownership of Bitcoins that you do not own. Someone must first transfer its ownership of the Bitcoins to you and only then can you transfer the ownership to others. That person must specify your Bitcoin address before transferring the Bitcoins. Now, this is not enough as then anyone intercepts or hacks this Bitcoin address will claim ownership of these Bitcoins. So, to ensure full security, the person who has the private key which created/owned the Bitcoin address, should be allowed to claim ownership of the Bitcoins.

The Bitcoin ecosystem decided to make this 'determination of ownership' its most valuable intellectual property. It is one the most important innovation in the technology behind Bitcoins in our view.

Mr. Satoshi created a scripting language and not a programming language to represent ownership or transfer of the Bitcoins. This language called by a very innovative name, Script has everything except for variables and loops. The experts say that it is not Turing complete.

Like the Forth programming language of the last century and the Microprocessors microcode, Script is stack based. A stack is an area of memory that has an imaginary stack pointer for moving upwards or downwards. Values, operands, parameters are pushed on the stack or memory and then functions or opcodes are called which look for them at the top of the stack for them. The stack must be cleaned up at the end of every function call.

The originator of these Bitcoins places the RipeMD hash and not the Bitcoin address of the new owners in the Output Script. This script contains other opcodes or programming language constructs as well. Since the code is executed in this scripting language, all protocols are to be strictly followed to claim ownership of these Bitcoins. All in all, the output contains a RipeMD hash plus more (script).

When a miner start validating a transaction, the input data in the input record goes first on the stack. The first value in this case is 72, so 72 bytes of the signature is pushed on the stack. This number may vary by 1 in both directions. The top of the stack now contains this signature. Then the script interpreter sees a value of 65, so it pushes the next 65 bytes of the uncompressed public key on the stack. The stack pointer points to the public key at the top of the stack, followed by the signature which is one below. The stack moves virtually down in memory, imagine like plates stacked one on top of the other. The stack, as of now, only contains data that will resolve the reference point in the transaction hash of the output.

The program or script is present in the Output section for the earlier transaction and not in the Input section of the current transaction. The input has the data and the earlier output has the rules written in a script. The input has the key, the lock is the output. The data supplied by the input can open the lock, as in disclose the true owner.

Let's look at the bytes in the `scriptPubKey` in the output section of the previous transaction. The first byte is the number

or opcode 0x76 which is OP_DUP or duplicate. It simply duplicates the value present on the top of the stack. Here, the public key is the last item on the stack, so it is duplicated. We now have two copies of the same uncompressed public key on the stack.

These are the instructions in the Output so we have no choice but to comply. To start with opcode OP_DUP is not a condition, it is a rule. These 25 bytes of the output have conditions, which are to be fulfilled. The output has protocols to follow, conditions returning true or false, and data. The original holder of the Bitcoin adds these rules which the receiver must comply to claim ownership. Similarly, the new owner writes his rules when transferring ownership to the next receiver.

Then follows the opcode 0xa9 or OP_HASH160. This value is displayed in the output. The Bitcoin wiki has names for every opcode. The public key present on top of the stack is passed through two algorithms, SHA-256 and RipeMD. The RipeMD hash value on the stack is 20 bytes long and starts with 46a, in our case. This value is also displayed in the output.

For this opcode, there were two public keys on the stack, now it is back to one. So, finally on the stack, there is a RipeMD hash newly created, the public key and finally the signature. The public key and signature remain unchanged.

Now the new opcodes are 0x14 or 20. This indicates that the next 20 bytes are to be placed on the stack. These 20 bytes are not the Bitcoin address in the output script but the RipeMD hash.

Any number from 1 to 75 is not really an opcode. It is an instruction to put the next set of bytes as per the number on the stack. It also means that the signature or public key is limited to 0x4b bytes only, as there is only one byte. For larger numbers on the stack, we have another chapter. So, the number 72 refers to placing 72 bytes on the stack. Thus, 72 as an opcode technically commands placing 72 bytes on the stack. At times, the inputs also contain code and just not data.

At this stage, there are two kinds of Bitcoin addresses on the stack, though technically they are not addresses, but RipeMD hash values.

Some opcode push data on the stack, some carry out the real work like calculating Bitcoin addresses.

The next opcode is 0x88 which is OP_EQUALVERIFY. This instruction checks if the top two items, that is the RipeMD hashes, on the stack are equal. If yes, they are popped off the stack and a value of true is returned. The stack finally contains only the public key and signature from the inputs.

The opcode 0xac or OP_CHECKSIG is of great importance as it calculates the signature, something we will work with in a later chapter. Signing and verifying transactions is the most difficult part of Bitcoins. Verifying a signature is basically proving that a unique private key signed the transaction, albeit without having the private key.

To summarize, so far all that we have done is simply checked if the RipeMD hash value of the public key in the input script is the same as the RipeMD hash value in the output the transaction hash points to. We are not looking at the output of the current transaction. We are looking at the current transaction's input and the output of the transaction this input is pointing to. The output of the transaction will be verified by the new owner, or when the new owner transfers ownership.

The RipeMD hash value is directly obtainable from the public key. The public key plays a major role in signature verification.

The question that arises here is why does an input carry the public key and the Output have a RipeMD hash value.

The script separates Bitcoins from earlier crypto currencies as it is a programmable currency. The code decides when and under what conditions, a Bitcoin owner is allowed to transfer ownership of his/her Bitcoins.

Here, the opcodes are deciphered in a very simple way. The assumption here is that there will only be certain opcodes present in the output. The bitcoin Wiki calls these types of transactions Pay to PubKey Hash.

This code is copied from the official Bitcoin Wiki.

```
scriptPubKey: OP_DUP OP_HASH160 <pubkeyhash> OP_EQUALVERIFY OP_CHECKSIG
scriptSig: <sig> <pubKey>
```

Finally, the Wiki is the last word. This output type is one of the simplest in the Bitcoin world. It is also one of the most common outputs seen in the blockchain.

Python and the `__name__` built in Global Variable

```
ch0803.py
print __name__
```

Output
`__main__`

Python has some zillion predefined variables that start with `__`. Just one of them being `__name__`. When we run the above program, Python gives this variable `__name__` a value of `__main__`.

This is confusing.

When to use the `from` Python Keyword

```
ch0804.py
from ch0803 import *
```

Output
`ch0803`

The difference between this program and the above one is that the python file `ch0803.py` is brought into `ch0804.py` using the `from` keyword. This is more like an import.

Now the same variable `__name__` gets a value of the Python file in which it is present `ch0803` and not the name `__main__`.

This feature is used generally in Python files having external functions. It determines which blocks of code get called when running standalone and which ones are being called using the `from` keyword.

Displaying the Output Script in a More Professional Manner

```
ch0805.py
scriptd = {
    0x00 : 'OP_FALSE',
    0x50 : 'OP_RESERVED',
    0x51 : 'OP_1',
    0x61 : 'NOP',
    0x62 : 'RESERVED OP_VER',
    0x63 : 'OP_IF',
    0x64 : 'OP_NOTIF',
    0x65 : 'RESERVED OP_VERIF',
    0x66 : 'RESERVED OP_VERIFNOTIF',
    0x67 : 'OP_ELSE',
```

0x68 : 'OP_ENDIF' ,
0x69 : 'OP_VERIFY' ,
0x6a : 'OP_RETURN' ,
0x6b : 'OP_TOTALSTACK' ,
0x6c : 'OP_FORMALSTACK' ,
0x6d: 'OP_2DROP' ,
0x6e: 'OP_2DUP' ,
0x6f: 'OP_3DUP' ,
0x70: 'OP_2OVER' ,
0x71: 'OP_2ROT' ,
0x72: 'OP_2SWAP' ,
0x73 : 'OP_IFDUP' ,
0x74 : 'OP_DEPTH' ,
0x75 : 'OP_DROP' ,
0x76 : 'OP_DUP' ,
0x77: 'OP_NIP' ,
0x78: 'OP_OVER' ,
0x79: 'OP_PICK' ,
0x7a: 'OP_ROLL' ,
0x7b: 'OP_ROT' ,
0x7c: 'OP_SWAP' ,
0x7d: 'OP_TUCK' ,
0x7e: 'OP_CAT' ,
0x7f: 'OP_SUBSTR' ,
0x80: 'OP_LEFT' ,
0x81: 'OP_RIGHT' ,
0x82: 'OP_SIZE' ,
0x83: 'OP_INVERT' ,
0x84: 'OP_AND' ,
0x85: 'OP_OR' ,
0x86: 'OP_XOR' ,
0x87: 'OP_EQUAL' ,
0x88 : 'OP_EQUALVERIFY' ,
0x89 : 'OP_RESERVED1' ,
0x8a : 'OP_RESERVED1' ,
0x8b: 'OP_1ADD' ,
0x8c: 'OP_1SUB' ,
0x8d: 'OP_2MUL' ,
0x8e: 'OP_2DIV' ,
0x8f: 'OP_NEGATE' ,
0x90: 'OP_ABS' ,
0x91: 'OP_NOT' ,
0x92: 'OP_0NOTEQUAL' ,
0x93: 'OP_ADD' ,
0x94: 'OP_SUB' ,
0x95: 'OP_MUL' ,
0x96: 'OP_DIV' ,

```

0x97:'OP_MOD',
0x98:'OP_LSHIFT',
0x99:'OP_RSHIFT',
0x9a:'OP_BOOLAND',
0x9b:'OP_BOOLOR',
0x9c:'OP_NUMEQUAL',
0x9d:'OP_NUMEQUALVERIFY',
0x9e:'OP_NUMNOTEQUAL',
0x9f:'OP_LESSTHAN',
0xa0:'OP_GREATERTHAN',
0xa1:'OP_LESSTHANOREQUAL',
0xa2:'OP_GREATERTHANOREQUAL',
0xa3:'OP_MIN',
0xa4:'OP_MAX',
0xa5:'OP_WITHIN',
0xa6:'OP_RIPEMD160',
0xa7:'OP_SHA1',
0xa8:'OP_SHA256',
0xa9 : 'OP_HASH160' ,
0xaa : 'OP_HASH256' ,
0xab : 'OP_CODESEPARATOR',
0xac : 'OP_CHECKSIG' ,
0xad:'OP_CHECKSIGVERIFY',
0xae:'OP_CHECKMULTISIG',
0xaf:'OP_CHECKMULTISIGVERIFY',
0xb0:'OP_IGNORED',
0xb1:'OPCHECKLOCKTIMEVERIFY',
0xb2:'OP_IGNORED',
0xb3:'OP_IGNORED',
0xb4:'OP_IGNORED',
0xb5:'OP_IGNORED',
0xb6:'OP_IGNORED',
0xb7:'OP_IGNORED',
0xb8:'OP_IGNORED',
0xb9:'OP_IGNORED',
0xfd:'INVALID OP_PUBKEYHASH',
0xfe:'INVALID OP_PUBKEY',
0xff:'INVALID OP_INVALIDCODE'
}
def fscript(script):
    script = script.decode('hex')
    i = 0
    ans = ""
    while i < len(script):
        byte = ord(script[i])
        if byte >=1 and byte <= 0x4b:

```

```
pkey = script[i+1 :byte + i + 1 ]
ans = ans + “ “ + pkey.encode('hex') + “ ‘ “
i = i + byte
else:
ans = ans + scriptd[byte] + ‘ ‘
i = i + 1
return ans

if __name__ == “__main__”:
a = fscript(“76a91446af3fb481837fadbb421727f9959c2d32a3682988ac”)
print a
```

Output

```
OP_DUP OP_HASH160 '46af3fb481837fadbb421727f9959c2d32a36829 '
OP_EQUALVERIFY OP_CHECKSIG
```

The Bitcoin Wiki gives us information on the value of every opcode and the name associated with it.

In the above program, we write a script engine which decodes all the possible opcodes.

There are some zillion combinations of opcodes also. In the program, we create a python dictionary called scriptd which has all the opcodes and the opcode string names. For example, the opcode 0xac is called OP_CHECKSIG.

A python dictionary is enclosed within {}. It has a key and a value which are separated by a :. The key and value can be of any data type.

The function fscript takes the encoded hex scriptPublicKey string as a parameter. We are hardcoding the same value we used in the last program. As mentioned earlier, encoded strings are for display purposes only. First, the string is decoded and the real string is saved in the variable script.

Though, we could have passed a decoded script but then that would not be visible to the naked eye.

We then enter a while loop as the script string has a variable length. Using a for or a while for looping is a personal choice. Then, the ord function converts every retrieved byte of the string, to an actual number. This value is saved in an aptly named variable called byte. The loop variable i acts as an index to the array as well.

Once the bytes are converted, we check if the variable byte represents an opcode. Just a reminder, that the opcode decides on the number of bytes to be pushed on the stack or if a signature is to be verified. Every opcode in Bitcoin has a value from 0 to 255 and it has a fixed job/role.

The only check we perform is if the opcode is less than 0x4b. If yes, then the slice operator is used to access the RipeMD hash value starting from the index position, i + 1. These offsets of +1 or -1 are very important.

The extraction of the RipeMD hash value needs some more understanding. When the value of i is 2, it points to byte 3 where the value is 0x14, (20), which is the length of the hash following.

These 20 bytes are stored in a variable called pkey. Since the length byte is not significant, the characters are extracted from position i + 1. Since we must extract 20 bytes, we add the value of the byte variable to the starting position i + 1. This gives an ending position of i + 1 + byte.

Another example, slice [4:7] reads 3 bytes, bytes 4 and 5 and 6. The answer remains the same when the end position is subtracted from the starting position.

In the if statement, we increase the loop variable i not by 1 but by the length of the string we have virtually removed. This length is stored in variable byte. After the if statement, the value of i is increased by 1. The if statement does not account for the length opcode, it is checked outside of the if statement as it applies to every byte read from the string.

The opcode may not always be a push data opcode. In that case, it is a 'doing some serious work' opcode. The string name is retrieved from the dictionary using the byte variable as the key. The return value is the opcode name in geeky English.

The program is an automated way to read and analyze a script string and it can work as a generic script engine. It's not complete.

We stop here because there are many types of transactions other than Pay to PubKey Hash that need attention. Finally, our goal is to write a program that will analyze every output of every Bitcoin transaction. You will be horrified to see the junk that is present in the outputs and what's more worrying is that the miners who mine the transactions perform no error checks.

CHAPTER 09

Hiding Data in the Blockchain

There are stories on the Internet which claim that the Bitcoin blockchain is used to store emails, pdf files, gif files and more. We initially thought it to be some sort of a prank. To our surprise, everyone acknowledges it today.

So, we started doubting ourselves, thinking we did not understand enough of the blockchain file format and decided to start from the beginning for the nth time. After all there is no place in the blockchain to hide an email, of all the things.

A bitcoin block starts with a magic number followed by the size of the data following, these values have a special meaning and hence cannot be changed.

Then there is the 80-byte block header that also has no space for storing a pdf file. None of these fields can be replaced in an arbitrary manner. The variable int following it gives a count on the number of transactions, which are stored back to back. Too small a value to be used but very important.

Let's look at the transaction data that takes up most of the space in a block file. Every transaction starts with an unused version field and ends with the locktime, which is rarely used. Between these two values, the inputs and outputs are sandwiched. The inputs take up more bytes than the outputs.

The inputs have 3 effective fields and two helper fields. It starts with a transaction hash and then the offset of the output in this transaction hash. The output resolves one half of the ownership puzzle. If this half is tampered with, no one can claim ownership of the Bitcoins. So, tampering with the transaction hash value is strictly a no-no, the same applies to the output index this hash points to.

Then comes the signature and public key as one field which is preceded with the length of the field. These values solve the other half of the ownership puzzle; they supply the data to confirm that the owner of the Bitcoins in the earlier output used his private key to sign the transaction. Both, the input and the output it points to, are a necessity. The output supplies the rules to be observed to claim ownership.

The owner of the Bitcoins creates the output which the receiver's input points to. The output is created in a manner that only the private key holder can claim ownership of the Bitcoin later.

If the owner of the Bitcoins does not want to transfer the ownership to the next person, it can place junk bytes in the script public key field of the output. The downside here is that nobody can spend these Bitcoins as the script would be effectively useless. The second half of the ownership puzzle would be compromised beyond repair.

It comes as no surprise then that the junk can only be in the output field. The RipeMD hash is only 20 bytes long. Any size of the hash can be fitted and nobody will complain. The complaints will come in when someone tried to redeem these bitcoins. If nobody does, the miners will not flag an error.

The question here is that why are the output scripts not verified for consistency?? After all there are less than a handful of possible script types that are found in the output. Is it because of a tearing hurry to be first that all error checks are ignored?

Let's give you real proof. The transaction hash:

77822fd6663c665104119cb7635352756dfc50da76a92d417ec1a12c518fad69

contains an email from the guru, Mr. Satoshi. You decide whether it is fake or not. The only saving grace is that the website blockchain.info agrees with us and calls this transaction strange. But it yet displays the email.

Let's see what the contents of this transaction are.

An Email Written by Mr. Satoshi Found in the Blockchain

```
ch0901.py
from cfuncs1 import *
from ch0805 import *
thash = '77822fd6663c665104119cb7635352756dfc50da76a92d417ec1a12c518fad69'
print "Transaction Hash is %s" % thash
(fno , inputa , outputa , locktime ) = tran(thash , 74)
for i in range (0 , len(inputa)):
    print "Input Number %d" % i
    print "\tTransaction Hash is %s" % inputa[i][0].encode('hex')
    print "\tOutput Index in transaction %x" % inputa[i][1]
    (fno1 , inputa1 , outputa1 , locktime1 ) = tran(inputa[i][0].encode('hex') , 74)
    for j in range (0 , len(outputa1)):
        if j == inputa[i][1]:
            print "\t\tOutput Number %d" % j
            print "\t\t\tValue of the Bitcoin %d:%.03f" % (outputa1[j][0], outputa1[j][0] * 1.0/100000000.00)
            print "\t\t\tLength of Output Script %d" % outputa1[j][1]
            print "\t\t\tOutput Public Key Script %s" % outputa1[j][2]

            print "\tInput Script Length ScriptSignature %d" % inputa[i][2]
            print "\tSignature + Public Key %s" % inputa[i][3]
            print "\tSequence Number %x" % inputa[i][4]

for i in range (0 , len(outputa)):
    print "Output Number %d" % i
    print "\tValue of the Bitcoin %d:%.03f" % (outputa[i][0], outputa[i][0] * 1.0/100000000.00)
    print "\tLength of Output Script %d" % outputa[i][1]
    print "\tOutput Public Key Script 30 bytes %s" % outputa[i][2][:30]
    print "\tOutput Public Key Script Last 2 bytes %s" % outputa[i][2][-2:]
    ans = ''
    for ch in outputa[i][2].decode('hex'):
        ans = ans + "%c" % ch
    print ans
```

Output

```
Transaction Hash is 77822fd6663c665104119cb7635352756dfc50da76a92d417ec1a12c518fad69
Input Number 0
Transaction Hash is 09847d68cfb15be6bda710647c006b0ef98d858b66422bd79e46ad5d80876c1e
Output Index in transaction 0
Output Number 0
Value of the Bitcoin 1900000:0.019
Length of Output Script 25
```

```
Output Public Key Script
76a914d9fc6bc120342fc43b2296876562e1d4c6536fda88ac
Input Script Length ScriptSignature 107
Signature + Public Key
48304502204f3353cf129bb805fb90315aeb6d5ab8e0937129c4b0f1422495e42b6bf0e9
28022100cbdd2811a4c94397aa731c224ee0c7290fea6bc17a9b3ce5957d9937c60f4f97
012103e2a0e6a91fa985ce4dda7f048fca5ec8264292aed9290594321aa53d37fdea32
Sequence Number ffffffff
Output Number 0
Value of the Bitcoin 1500000:0.015
Length of Output Script 1455
Output Public Key Script 30 bytes 63ff054effffffff4da30546726f6d
Output Public Key Script Last 2 bytes 68
Opcodes OP_IF INVALID OP_INVALIDCODE '4effffffff' 'a30546726f6d'

c?N????M?From a3a61fef43309b9fb23225df7910b03afc5465b9 Mon Sep 17 00:00:00 2001
From: Satoshi Nakamoto <satoshin@gmx.com>
Date: Mon, 12 Aug 2013 02:28:02 -0200
Subject: [PATCH] Remove (SINGLE|DOUBLE)BYTE

I removed this from Bitcoin in f1e1fb4bdef878c8fc1564fa418d44e7541a7e83
in Sept 7, 2010, almost three years ago. Be warned that I have not
tested this patch.

—
backends/bitcoind/deserialize.py | 8 +----
1 file changed, 1 insertion(+), 7 deletions(-)

diff --git a/backends/bitcoind/deserialize.py b/backends/bitcoind/deserialize.py
index 6620583..89b9b1b 100644
--- a/backends/bitcoind/deserialize.py
+++ b/backends/bitcoind/deserialize.py
@@ -280,10 +280,8 @@ opcodes = Enumeration("Opcodes", [
    "OP_WITHIN", "OP_RIPEMD160", "OP_SHA1", "OP_SHA256", "OP_HASH160",
    "OP_HASH256", "OP_CODESEPARATOR", "OP_CHECKSIG", "OP_CHECKSIGVERIFY", "OP_CHECKMULTISIG",
    "OP_CHECKMULTISIGVERIFY",
- ("OP_SINGLEBYTE_END", 0xF0),
- ("OP_DOUBLEBYTE_BEGIN", 0xF000),
    "OP_PUBKEY", "OP_PUBKEYHASH",
- ("OP_INVALIDOPCODE", 0xFFFF),
+ ("OP_INVALIDOPCODE", 0xFF),
])

@@ -293,10 +291,6 @@ def script_GetOp(bytes):
    vch = None
    opcode = ord(bytes[i])
    i += 1
- if opcode >= opcodes.OP_SINGLEBYTE_END and i < len(bytes):
- opcode <= 8
- opcode |= ord(bytes[i])
```

```

- i += 1
    if opcode <= opcodes.OP_PUSHDATA4:
        nSize = opcode
-
1.7.9.4
h
c?N???M?F

```

The output displays 'the Mr. Satoshi email' in all its glory. This email is a permanent part of the blockchain.

Let's understand the code that displays this email. In the process, it also answers some pertinent questions like, why did the miners allow such a transaction to be part of the blockchain.

This example has very few new lines of code, most of it has been recycled from the past. The same variable `thash` stores the transaction hash 7782. It must be noted that this transaction hash contains the hidden email. The function `tran` found in the file `cfuns1.py`, as before, returns the transaction data when it is given the hash and the starting block number. The block number should be the largest block number from which the search takes place backwards. Since we are aware that the offending block no is 74, we hardcode it. You please use a large block number like 535 and wait for the cows to come home.

The latter part of the code simply displays the transaction data. We are displaying the inputs which can be dismissed. As we mentioned earlier, ignore the inputs and focus only on the outputs.

First anomaly here is that the length of the output script is a whopping 1455 bytes. Compare this with a valid script seen in the past, which is roughly 25 bytes; 20 bytes are for the RipeMD hash value and then some standard opcodes.

The first byte is 0x63 which is the opcode for the script, if statement `OP_IF`. The if statement here is like all if statements, if the condition on stack is true then the following series of statements are executed. If false then the statements after opcode 0x67 `OP_ELSE` gets executed. The last byte should be a 0x68, the `OP_ENDIF`.

The rules are very clear, if the transaction has a `OP_IF` opcode, then it must also have a `OP_ENDIF` or the transaction is marked invalid. You can see the `OP_IF` and `OP_ENDIF` better in the blockchain browser. The first byte is the opcode `OP_IF` or 0x63.

Thus, this output script starts with an if and ends with an endif. As of now, it is a mystery as to what is to be checked on the stack. The verifier also does not go into these details, all that it checks for, is a valid transaction with an if and endif and it passes the code. We will never know whether the if statement resulted in true or false.

Interestingly, we see an opcode 0xff which is aptly called `OP_INVALIDOPCODE`. The guide says this opcode matches any opcode that is not yet assigned, though we are clueless as to what it means. This is the second byte, 0xff. Since the next opcode is 5, five bytes are put on the stack.

The rest of the code makes no sense to anyone and it will never be executed by anyone.

We are only displaying the first 30 bytes of this string using the slice operator `[:30]`.

Then, using `[-2:]`, (the -2 means the last 2, the empty `:` means till the end) the last 2 bytes are extracted. It is the `OP_ENDIF` opcode. No point displaying all the 1455 junk bytes.

The blockchain.info site discloses that the inputs total is 0.019 Bitcoins and the outputs is 0.015, the difference is the fee that goes to the miner, is 0.004 Bitcoins.

To display the actual email, the entire encoded string is decoded. In our function, there is a for loop wherein every byte

is added to a string ans in a character form, using the modifier %c. The real email starts from the 12th byte This is where the word From starts. The initial bytes are to satisfy the if statement and the last byte is the endif.

As an aside, if you skip the first 9 bytes, we will see two bytes a3 and 05, in decimal these are 163 and 5. Converting this short into a number gives a value of $5 * 256 = 1280 + 163 = 1443$ plus 12 (the initial bytes as per the last for statement) which is the size of the output string, 1445. Not a coincidence !!!

Also, it is a recent block, the date is 12-August-2013 12:32:11, 4 years back.

Modified tran Function

```
cfuncs1d.py
def tran(phash , start):
    for fno in range(start , -1 , -1):
        fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat" % fno
        #print fname
        try:
            f = open(fname , "rb")
        except:
            break
        while (True):
            try:
                magic = rint(f)
                if magic == 0:
                    continue
            except:
                break
            size = rint(f)
            header = f.read(80)
            notran = rvarint(f)
            for tno in range ( 0 , notran):
                startfp = f.tell()
                tver = rint(f)
                inputa = dinput(f )
                outputa = doutput(f)
                locktime = rint(f)
                endfp = f.tell()
                f.seek(- (endfp - startfp) , 1)
                trand = f.read(endfp - startfp)
                thash = chash(trand)
                thash = thash[::-1].encode('hex')
                if thash == phash:
                    return (fno , inputa , outputa , locktime , trand)
            return (-1 , [], [], 0 , [])
```

This example extracts the Satoshi pdf file on Bitcoins which is stored as part of a transaction. This file is the gold standard that describes the internals of the Bitcoin protocols. The pdf file is 180K large and obviously, it cannot fit into one output. It needs to be stored in more than output of a single transaction.

The offending transaction hash is:

54e48e5f5c656b26c3bca14a8c95aa583d07ebe84dde3b7dd4a78f4e4186e713.

It is in file blk00052.dat. Even the blockchain browser cannot decipher this transaction, but the miners passed it.

The same function tran of the previous program is slightly modified and placed in a file called cfuncs1d.py. The other two functions doutput and dinput are the same, so they are not displayed here.

The function now returns an extra parameter. The transaction bytes are read from disk and stored in the variable trand. These raw transaction bytes are then used to calculate the transaction hash.

The other change made in the program is that the return statement now returns 5 values. The new value is given to the variable trand, where the entire transaction data is stored, as we just mentioned.

Reading the original Satoshi Bitcoin paper as a pdf file stored in the blockchain

```
ch0902.py
from cfuncs1d import *
from ch0805 import *
import os
thash = '54e48e5f5c656b26c3bca14a8c95aa583d07ebe84dde3b7dd4a78f4e4186e713'
print "Transaction Hash is %s" % thash
(fno , inputa , outputa , locktime , trand) = tran(thash , 52)
for i in range (0 , len(inputa)):
    print "Input Number %d" % i
    print "\tTransaction Hash is %s" % inputa[i][0].encode('hex')
    print "\tOutput Index in transaction %x" % inputa[i][1]
    print "\tInput Script Length ScriptSignature %d" % inputa[i][2]
    print "\tSignature + Public Key %s" % inputa[i][3]
    print "\tSequence Number %x" % inputa[i][4]
    (fno1, inputa1, outputa1, locktime1, trand1) = tran(inputa[i][0].encode('hex'), 52)
    for j in range (0 , len(outputa1)):
        if j == inputa[i][1]:
            print "\t\tOutput Number %d" % j
            print "\t\t\tValue of the Bitcoin %d:%.05f" % (outputa1[j][0], outputa1[j][0] * 1.0/100000000.00)
            print "\t\t\tLength of Output Script %d" % outputa1[j][1]
            print "\t\t\tOutput Public Key Script %s" % outputa1[j][2]
print "Number of Outputs %d" % len(outputa)
for i in range (0 , len(outputa) - 945 ):
    print "Output Number %d" % i
    print "\tValue of the Bitcoin %08d:%.08f" % (outputa[i][0], outputa[i][0] * 1.0/100000000.00)
    print "\tLength of Output Script %d" % outputa[i][1]
    print "\tOutput Public Key Script 30 bytes %s" % outputa[i][2][:30]
    print "\tOutput Public Key Script Last 2 bytes %s" % outputa[i][2][-2:]
    ans = fscript(outputa[i][2][:30])
    print "\tOpcodes %s" % ans
    print
os.system("rm bitcoin.pdf zzz.dat")
ans = ""
```

```
for ch in outputa[0][2].decode('hex'):
    if ord(ch) >= 32 and ord(ch) <= 127:
        ans = ans + "%c" % ch
print ans
f = open("zzz.dat", "wb")
f.write(trand)
f.close()
print "Length of original Transaction Data %d" % len(trand)
raw = trand.encode('hex')
outputs = raw.split("0100000000000000")
print "Length of the Outputs %d" % len(outputs)
print "0:%d 1:%d 2:%d n-2th:%d n-1th:%d nth:%d" % (len(outputs[0]), len(outputs[1]), len(outputs[2]),
len(outputs[len(outputs) - 3]), len(outputs[len(outputs) - 2]), len(outputs[len(outputs) - 1]))
f = open("zzz0.dat", "wb")
f.write(outputs[0].decode('hex'))
f.close()
f = open("zzz1.dat", "wb")
f.write(outputs[1].decode('hex'))
f.close()
f = open("zzz2.dat", "wb")
f.write(outputs[2].decode('hex'))
f.close()
f = open("zzz3.dat", "wb")
f.write(outputs[-2][6:-4].decode("hex"))
f.close()
a = [ 11 , 12 , 13 , 14 , 15 , 16 , 17 , 18]
print a[1:-2]
print a[-2]
print "Length of the outputs we use %d" % len(outputs[1:-2])
ans = "A"
ans += "B"
ans += "C"
print ans
pdf = ""
for output in outputs[1:-2]:
    cur = 6
    pdf += output[cur:cur+130].decode('hex')
    cur += 132
    pdf += output[cur:cur+130].decode('hex')
    cur += 132
    pdf += output[cur:cur+130].decode('hex')
pdf += outputs[-2][6:-4].decode("hex")
f = open("bitcoin.pdf", "wb")
f.write(pdf[8:])
f.close()
os.system("open bitcoin.pdf")
```

Output

Transaction Hash is

54e48e5f5c656b26c3bca14a8c95aa583d07ebe84dde3b7dd4a78f4e4186e713

Input Number 0

Transaction Hash is

6c53cd987119ef797d5adccd76241247988a0a5ef783572a9972e7371c5fb0cc

Output Index in transaction 7

Input Script Length ScriptSignature 107

Signature + Public Key

483045022012410c1f4b53cc2eae98e1d41bfa3569cce8ddb7e76814e4b4c9195129f3e5
 d0022100fd9e49f15f8aea02e09909cf73eab78d0a8f2cd18a5cb2626124036fe0c420b20
 121024d1249ddcb03c6edec4ab0ea77f379910dbb20e9b1404d5098bbaa9e748a6bd8

Sequence Number ffffffff

Output Number 7

Value of the Bitcoin 78305768:0.78306

Length of Output Script 25

Output Public Key Script

76a9147922cbce6b2c1206c687f486ed7afa55e73ea01488ac

Number of Outputs 948

Output Number 0

Value of the Bitcoin 00000001:0.00000001

Length of Output Script 201

Output Public Key Script 30 bytes 5141e4cf0200067daf13255044462d

Output Public Key Script Last 2 bytes ae

Opcodes OP_1 'e4cf0200067daf13255044462d '

Output Number 1

Value of the Bitcoin 00000001:0.00000001

Length of Output Script 201

Output Public Key Script 30 bytes 5141f4eb5fde17f7b8fbf7c53feeee

Output Public Key Script Last 2 bytes ae

Opcodes OP_1 'f4eb5fde17f7b8fbf7c53feeee '

Output Number 2

Value of the Bitcoin 00000001:0.00000001

Length of Output Script 201

Output Public Key Script 30 bytes 5141b785538d5199a8781eee9f7dc5

Output Public Key Script Last 2 bytes ae

Opcodes OP_1 'b785538d5199a8781eee9f7dc5 '

QA}%PDF-1.4%2 0 obj<</Length 3 0

R/Filter/FlateDeAcode>>streamx\K\$))%eBSSsma%[Rf]AG(j7w})?e|{ni[\S

Length of original Transaction Data 198724

Length of the Outputs 948

0:312 1:404 2:404 n-2th:404 n-1th:76 nth:128

[12, 13, 14, 15, 16]

17

Length of the outputs we use 945

ABC

The from statement has the file cfuncs1d in place of cfuncs. This file brings in the modified function, tran. Also, in the program, some external programs are used through the command line, so the os module is also included.

The tran functions as we mentioned earlier returns 5 values. There is only one reason to display the output referenced by the input. It is to disclose that the output only has very few Bitcoins to offer; less than 1 Bitcoin. It's too expensive today to use one Bitcoin on nothing.

The input looks normal as there can be no useless data ever stored there, its only stored in the outputs. The reason being, the output is someone else's problem.

The reason behind hiding stuff in the blockchain is very simple. The code in the script of the output is only executed when someone claims ownership of those Bitcoins. Again, the output script code is checked if and only if someone claims ownership of these bitcoins otherwise this code is never executed. The world takes advantage of this ambiguity. Since a miner does not check any code while mining a transaction, any junk can be easily added in the output script code.

The length of the signature and public key is a measly 107 bytes long. Let's move on to the main culprit, the outputs.

The problem is in the sheer number of outputs, 948 of them, all in just one transaction. It should ring alarm bells someplace as 0.7 bitcoins are distributed to a whopping 948 bitcoin addresses. Each output however looks the same. The bitcoin output has a value of 1 Satoshi or 0.00000001, a very small number and the length of the output script length is 201, not a standard value. We are printing only the first 3 outputs but it shows some pattern.

The script starts with the opcode 51 which is named as OP_1. It pushes the number 1 on the stack. The opcode 41 pushed the subsequent 0x41 bytes on the stack.

The function, system from the os module is used to run any os command line program from python code. The function takes a string which is the command generally typed in the command window.

First, the file bitcoin.pdf is deleted using the rm command. The objective here is to create a pdf file called bitcoin.pdf with the transaction data saved in variable trand. So, old file is deleted.

The variable trand contains all the bytes of the transaction. It cannot be used directly as it consists supplementary Bitcoin data along with a pdf file formatted data, all thrown in together.

The pdf file format is, in our humble opinion, is the most convoluted file format ever created by humanity. We have studied zillions of file formats and the pdf file format is the most complicated of all. A PDF files starts with reserved words %PDF and ends with %EOF. Every file has its own magic number to indicate start and sometimes, end a file.

The transaction data follows the pdf file format but this data is spread over multiple outputs. So, all data from different outputs is extracted and then affixed to create a valid pdf file. The length of transaction data is large.

The decoded variable trand is saved into an encoded variable raw. We display a few ascii bytes of a output which clearly shows the starting magic numbers %PDF and more.

The size of the original transaction data is a whopping 198724 bytes and the final pdf file is 184300 bytes large. Writing the transaction bytes to disk in file zzz.dat is only to gauge the size of the original data. The file zzz.dat is not used in the code ever.

The transaction data variable raw is of string type. The split function is used to split the string on a pattern which is 01 followed by 14 0's or 7 bytes. All the transaction headers are skipped to arrive at the bitcoin value. This string pattern is like a treasure clue.

A part of the pdf file is stored in each output script key. It seems that the owner did not want a very large output.

All outputs start with the same bitcoin value. This helps in breaking up our transaction output at the bitcoin value. This value is stored in 8 bytes as 0100000000000000. Again, we break up the Bitcoin transaction where the Bitcoin value

starts. For the less focused, the Bitcoin value is the first field in the outputs. This approach is put in place after observing how a Bitcoin value of 1 Satoshi is stored on disk.

The only available place to store any extra data is after the Bitcoin value. However, after this value field, is the size of the script public key data. The size of the list variable called outputs is displayed and it shows 948 which matches the count value or the number of outputs. An uncanny coincidence. So, there are 948 different Bitcoin values or outputs.

Next, the sizes of the first three split strings are displayed. The length of the first output [0] is 312 but the second and third one have a constant, 404 bytes. This is the size of an encoded string which is nearly double the length of Output Script length of 201. The magic comes in the last three output sizes. The third last is 404. However, the size of the last two outputs are 76 and 128. When working with these split generated outputs, we ignore the last two list members.

For the purposes of understanding these strings better, the first three outputs are written to disk. The decoded values are written because transaction data on disk is always decoded. The file zzz0.dat contains the data from list output[0], zzz1.dat from list outputs[1] and so on and so forth. None of the outputs start with the Bitcoin value 01000000; this string pattern is eliminated. Please open all these zzz files in your favorite hex editor and look at the bytes to verify our claim.

File zzz0.dat has no %PDF, however, the file zzz1.dat contains the magic numbers %PDF, but after some bytes. Therefore, we must start the reading from the pdf file from output[1] and not output[0]. We ignore the first list member. Our final pdf file, bitcoin.pdf must start with the bytes %PDF.

For the record, when we printed the opcodes earlier, it reconfirmed the fact that after the Bitcoin value were the magic characters, %PDF. In the file zzz1.dat, the magic words start at a certain offset. Also, the last couple of bytes of zzz.dat have the end of pdf marker %%EOF someplace at the end of this file. The file zzz.dat has the transaction data.

The last file zzz3.dat contains the words startxref. As per the pdf file format, the keyword points to the byte offset from where one can start reading the pdf file. Then comes the end magic marker, %%EOF. The slice operator [-2] gives the second last member of the list. This member marks the end, so we stop searching for any more pdf data.

Let's take a primer on slices. The list called a consists of 8 members or numbers.

The slice operator [1:-2] indicates the following. '1:' means start from the first and not the 0th member. Thus, it ignores the number 11. '-2' is telling the slice operator to stop extracting at the second last member, 16. The members with value 17 and 18 are therefore ignored. The operator += concatenates the strings by adding then at the end only.

The last 2 list members are ignored as the size is not consistent 404. The first output member is also ignored as it does not contain the %PDF magic number. The list member outputs[1:-2] follows the above rules. End of primer.

The code that follows is obviously copied from the person who added the pdf file to the blockchain. All the above code can be ignored.

The first 6 bytes are skipped using the variable cur. The next 130 characters or 65 bytes are read and concatenated into the string pdf. The next 132 characters are skipped by increasing the variable cur by 132. Then, again, the next 130/65 bytes are read and the value of variable cur is increased by 132.

The first offset is output[6:136]. The next output is [138:268] and finally output[270:400]. There is a gap of two characters between each read. The pdf string is decoded after each read. Finally, a part of the second last output tuple member is read, which is the position of the end marker %EOF.

f.write(pdf[9:]) : If the value 9 is changed to 8 in the last write function, the pdf file fails to open. Nevertheless, when the pdf file is opened, the first % of the %PDF is not seen.

Fortunately, we have learnt pdf file format but we are not smart enough to write code like this. Though, we would like to write a book on how miners do not perform any sanity checks on a transaction. Anything can pass for a valid transaction.

As an exercise, use our earlier code and scan for outputs whose script length exceeds 200 or where the number of outputs is very large. Lots of people on the internet have already done this and you could read all about the transaction hashes that have various types of contents like images etc. This list is endless.

The Bitcoin protocol allows the use of the opcode `OP_RETURN` to store any data that you want in the blockchain. The Bitcoin Wiki mentions certain valid transactions outputs, please stick only to them as there are many outputs that do not make any sense at all.

CHAPTER 10

Signing Transactions

In this chapter, we keep our promise and show you how a Bitcoin signature is verified. We will be using data from a Bitcoin transaction.

One of the programs in the Bitcoins chapter displayed a private key beginning with 420. From this private key, a public key starting with 04f9 was generated. Then, we explained that we were no mathematicians or cryptographers, so would not attempt to explain this magic till some time elapses. Let's understand the magic in this chapter.

Verifying the Signature on a Simple Message

```
ch1001.py
import pybitcointools
privkey = '4200000000000000000000000000000000000000000000000000000000000000'
print "Private Key %d" % (len(privkey) )
pubkey = pybitcointools.privtopub(privkey)
print "Public Key %s:%d" % ( pubkey , len(pubkey))
message = "Vijay"
sig = pybitcointools.ecdsa_sign(message , privkey)
print "Signature is %s:%d" % ( sig , len(sig))
ans = pybitcointools.ecdsa_verify(message , sig , pubkey)
print "Signature Verification      %r" % ans

ans = pybitcointools.ecdsa_verify(message , sig , privkey)
print "Signature Verification Switched  %r" % ans

ans = pybitcointools.ecdsa_verify("Vljay", sig , pubkey)
print "Signature Verification Message  %r" % ans

pubkey =
'04f9a7699db2eec6a4116373fde8e5c7f46af3d81c96b912cdc0e2dd67d00ed6d7081888
bc7c39617d4f7c3437b4ace461402174ed76561090838123f952d41489'
ans = pybitcointools.ecdsa_verify(message , sig , pubkey)
print "Signature Verification PublicKey %r" % ans
```

Output

```
Private Key 64
Public Key
04f9a7699db2eec6a4116373fde8e5c7f46af3d81c96b912cdc0e2dd67d00ed6d7081888
bc7c39617d4f7c3437b4ace461402174ed76561090838123f952d41488:130
Signature is
```

```
HP6fKUkn+t2smUe+w5bXlYnC87j3JAyeO8LiYJt6d4NJG0zpHwnBtXoN//  
Rds93BsQYdLy6JpuxmSRUbcXmTkOc=:88  
Signature Verification      True  
Signature Verification Switched False  
Signature Verification Message False  
Signature Verification PublicKey False
```

Let's now sign a message 'Vijay' which is my name. A little later in the chapter, we will sign a real Bitcoin transaction.

A function called `ecdsa_sign` is used from the module `pybitcointools`. This function requires a private key and the message or text or data to be signed.

This function returns a signature that starts with HP6f. Once again, we have no idea, at present, how this magic works. All that we know is that the private key and the message generates this signature. It is obvious, that changing the private key and/or the message will change the signature also. The new signature depends upon two factors, the actual message to be signed and the private key.

So, how is this any different from a hash? For starters, a hash does not require a private key. A hash uniquely identifies some data, a signature on the other hand uniquely identifies a combination of data and a private key.

To verify that the signature has been created by a certain private key, a function called `ecdsa_verify` is used. This function takes 3 parameters. The first two parameters are the original message and the signature. The last parameter is most important. It is not the private key but the public key. If the function returns `True`, it is confirmed that the private key has signed the message.

Once again, the only common entity in both, signing and verification, is the message. The private key is not shared with anyone. The public key derived from the private key is given to the public and it is used to verify the origins of the signature. This process is one way only. Thus, in the verification process, the signature, the message and public key are shared. A value of `true` reaffirms that the message is signed by a certain private key. Only the private key can generate its equivalent public key. The private key remains private forever.

If the last parameter, the public key, of the verify function is replaced with the private key, the function returns `False`. This means that only the public key which is derived from the private key can verify the signature.

In the next call of the `ecdsa_verify` function, the message is changed, the `i` is capitalized. The verification obviously fails. Finally, the last byte of the original public key is changed from 8 to 9 and once again the verification fails.

So, to summarize, first some text or Bitcoin transaction is signed with a private key. This private key has an equivalent public key. Only this public key can verify the transaction. To achieve this, the function is given the same message, the signature generated from the private key + message and the public key. Here, you must bear in mind that this private key is also used to generate the Bitcoin address.

If the signature verify function returns `true`, it can be safely concluded that the private key owner owns the Bitcoins. Once the ownership is proved, these Bitcoins can be transferred to the next owner.

We would like to verify the signature of only one specific transaction which is `cca7507897abc89628f450e8b1e0c6fca4ec3f7b34cccf55f3f531c659ff4d79` present in block number 57044 or blk file 0. This transaction was displayed in the output of program `ch0801.py`. Let's check if the signature was verified correctly.

Output of Program ch0801.py

Output

Transaction Hash is

`cca7507897abc89628f450e8b1e0c6fca4ec3f7b34cccf55f3f531c659ff4d79`

```

Input Number 0
Transaction Hash is
a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d
Output Index in transaction 0
Output Number 0
Value of the Bitcoin 1000000000000:10000.00
Length of Output Script 25
Output Public Key Script
76a91446af3fb481837fadbb421727f9959c2d32a3682988ac

Input Script Length ScriptSignature 139
Signature + Public Key
4830450221009908144ca6539e09512b9295c8a27050d478fbb96f8addbc3d075544dc41
328702201aa528be2b907d316d2da068dd9eb1e23243d97e444d59290d2fddf25269ee0
e0141042e930f39ba62c6534ee98ed20ca98959d34aa9e057cda01cfd422c6bab3667b76
426529382c23f42b9b08d7832d4fee1d6b437a8526e59667ce9c4e9dcebcabb
Sequence Number ffffffff

```

We have chosen this specific transaction on purpose as it has one input only. Too many inputs spoil the broth. To verify a signature, the input is most important. The outputs of this transaction are of no use. However, the output that is significant here, is the output referenced by the inputs of this transaction hash value.

Extracting the Transaction Hash and Output Index from the Inputs of a Transaction

```

ch1002.py
import pybitcointools
from cfuncs1d import *
thash = 'cca7507897abc89628f450e8b1e0c6fca4ec3f7b34cccf55f3f531c659ff4d79'
(fno , inputa , outputa , locktime , trand) = tran(thash , 0)
strd = pybitcointools.deserialize(trand.encode('hex'))
print type(strd)
print strd
print
print type(strd['ins'])
print strd['ins']
print
print type(strd['ins'][0])
print strd['ins'][0]
print
print type(strd['ins'][0]['outpoint'])
print strd['ins'][0]['outpoint']
print
prevhash = strd['ins'][0]['outpoint']['hash']
outputindex = strd['ins'][0]['outpoint']['index']
print "Previous Hash bearing Bitcoins %s" % prevhash
print "Output index in above hash %d" % outputindex

```

Output

```
<type 'dict'>
{'locktime': 0, 'outs': [{'value': 577700000000, 'script':
'76a914df1bd49a6c9e34dfa8631f2c54cf39986027501b88ac'}, {'value': 422300000000,
'script':
'4104cd5e9726e6afeae357b1806be25a4c3d3811775835d235417ea746b7db9eeab33cf
01674b944c64561ce3388fa1abd0fa88b06c44ce81e2234aa70fe578d455dac'}]},
'version': 1, 'ins': [{'script':
'4830450221009908144ca6539e09512b9295c8a27050d478fbb96f8adddbc3d075544dc4
1328702201aa528be2b907d316d2da068dd9eb1e23243d97e444d59290d2fddf25269ee
0e0141042e930f39ba62c6534ee98ed20ca98959d34aa9e057cda01cfd422c6bab3667b7
76426529382c23f42b9b08d7832d4fee1d6b437a8526e59667ce9c4e9dcebcabb',
'outpoint': {'index': 0, 'hash':
'a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d'},
'sequence': 4294967295}]}

<type 'list'>
[{'script':
'4830450221009908144ca6539e09512b9295c8a27050d478fbb96f8adddbc3d075544dc4
1328702201aa528be2b907d316d2da068dd9eb1e23243d97e444d59290d2fddf25269ee
0e0141042e930f39ba62c6534ee98ed20ca98959d34aa9e057cda01cfd422c6bab3667b7
6426529382c23f42b9b08d7832d4fee1d6b437a8526e59667ce9c4e9dcebcabb',
'outpoint': {'index': 0, 'hash':
'a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d'},
'sequence': 4294967295}]

<type 'dict'>
{'script':
'4830450221009908144ca6539e09512b9295c8a27050d478fbb96f8adddbc3d075544dc4
1328702201aa528be2b907d316d2da068dd9eb1e23243d97e444d59290d2fddf25269ee
0e0141042e930f39ba62c6534ee98ed20ca98959d34aa9e057cda01cfd422c6bab3667b7
6426529382c23f42b9b08d7832d4fee1d6b437a8526e59667ce9c4e9dcebcabb',
'outpoint': {'index': 0, 'hash':
'a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d'},
'sequence': 4294967295}

<type 'dict'>
{'index': 0, 'hash':
'a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d'}

Previous Hash bearing Bitcoins
a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d
Output index in above hash 0
```

We are reusing code written in the earlier chapters. The function called `tran` in the file `cfuns1d`, returns not only a list with the inputs and outputs but also returns all the bytes that make up the transaction. These transaction bytes are stored in the string variable `trand` as always, and in its present form, they have no structure, no meaning.

The `pybitcointools` has a function called `deserialize` that accepts a string of bytes and then interprets them as bytes of a real Bitcoin transaction. The job of this function is to figure out the structure of a transaction. This `deserialize` function

breaks up the transaction into tuples of inputs and outputs. Plus, it goes one step further and simplifies data access. The data type returned by the `deserialize` function is a dictionary. The resultant dictionary looks all messed up when printed, unless you like reading a Python dictionary filled with key-value pairs. The fields look completely out of order. The field `locktime`, which is supposed to be the last field in the transaction structure is displayed first. The field `locktime` has a value of 0 as always, but times are also known to change. If the fourth last byte of the variable `trand` is changed to 01000000, the `locktime` shows a value of 1. The default output is shown in reverse order.

The key `ins` in the dictionary represents all the inputs in the transaction. It would make a lot of sense if it was called `inputs` instead of a cryptic `ins`. So, we look for the key called `ins` in the dictionary. This key is followed by the colon, `:`. As everything is backwards, the `locktime` key comes first, it is followed by the `outs` key and then comes the `ins` key. The `ins` key has a list.

As learnt earlier, a list has multiple members or values. Every list starts with a `[` and ends with a `]`, in between these `[]`, multiple values are present. In our unique case, it has one value as there is one input only.

We chose a transaction with one input, as it is easier to verify the signature. Multiple inputs can get confusing. Outputs once again, are insignificant for now.

The transaction details present in the dictionary `strd` are first printed. The variable `strd['ins']` prints the list members of the `ins` key. This data type is a list and not a dictionary. To access the first and the only member of the list, the `[]` notation is used. The variable is therefore `strd['ins'][0]`. An index of 0 returns a dictionary, which is multiple keys and values.

Now, the next important key is called `outpoint` which is once again a dictionary representing the two key values. The first is a transaction hash value that contains the output. The second is an output index within this transaction. The variable used is `strd['ins'][0]['outpoint']`.

A more humane way of working with transactions.

It seems that the Bitcoin gods had decided in their infinite wisdom, that the transaction hash value and the output index come last. The `outpoint` dictionary is sandwiched between the script and sequence number. Since the script signature and public key are available, the script length field is not required. Every string carries its length with it.

Finally, the two members of the final dictionary are accessed using the key names, `hash` and `index`. The variables used are `strd['ins'][0]['outpoint']['hash']` and `strd['ins'][0]['outpoint']['index']`.

This way, we have extracted successfully, the previous transaction hash and output index from the transaction bytes.

Extracting the Signature and Public Key from the Inputs

```
ch1003.py
import pybitcointools
from cfuncs1d import *
thash = 'cca7507897abc89628f450e8b1e0c6fca4ec3f7b34ccccf55f3f531c659ff4d79'
(fno, inputa, outputa, locktime, trand) = tran(thash, 0)
strd = pybitcointools.deserialize(trand.encode('hex'))
phash = strd['ins'][0]['outpoint']['hash']
outi = strd['ins'][0]['outpoint']['index']
print "Previous Hash %s" % phash
print "Output index %d" % outi
scr = strd['ins'][0]['script']
scrd = scr.decode('hex')
siglen = ord(scrd[0])
```

```
    siga = scrd[1:siglen+1]
    sige = siga.encode('hex')
    pkeylen = ord(scrd[siglen + 1])
    pkey = scrd[siglen+2: siglen+1+ pkeylen+1]
    pub = pkey.encode('hex')
    print "Public Key is %s" % pub
    print "Signature is %s" % sige
```

Output

Previous Hash

a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d

Output index 0

Public Key is

042e930f39ba62c6534ee98ed20ca98959d34aa9e057cda01cfd422c6bab3667b7642652
9382c23f42b9b08d7832d4fee1d6b437a8526e59667ce9c4e9dcebcabb

Signature is

30450221009908144ca6539e09512b9295c8a27050d478fbb96f8addbc3d075544dc4132
8702201aa528be2b907d316d2da068dd9eb1e23243d97e444d59290d2fddf25269ee0e0

1

The program output displays data that looks Greek and Latin. In the last program, the transaction hash and the output index were disclosed, here we access the scriptsig or the signature and public key stored in the input.

The objective is to check if the transaction starting with hash value cca75 has been properly signed or not. For this purpose, the private key that signed the transaction is required, which will then give the computed signature and the public key. We will never ever have access to this private key.

As before, the script field is of prime importance here. This field has the signature and the public key. So, we first access the ins list, then a 0, which is the index in this list and lastly the key, script. The script's value is decoded and stored in the variable scr to obtain the original data consisting of the signature and public key. No error checks or exceptions are placed in the code as our assumption is that there will be a signature and a public key following it.

The encoded string, scr is decoded into variable scrd as always. The first byte of the string gives the length of the signature. The ord function is used and the return value is stored in the variable siglen.

Using the slice operator with 1 in siglen plus 1, the actual decoded signature is obtained and stored in variable siga. To display the value in variable siga, it is encoded and saved in variable sige. The value in sige is displayed though it makes no sense.

Now let's extract the public key. The public key value is stored after the entire signature. The variable siglen plus 1 is the length of the signature. The ord function saves the length in variable pkeylen.

The extracting of public key is a little more convoluted. The offset starts 2 bytes after the signature length as there are two length variables. It must be noted that there are two length bytes plus the lengths of the signatures themselves.

Verifying a Real Bitcoin Transaction's Signature

```
ch1004.py
import pybitcointools
from cfuncs1d import *
thash = 'cca7507897abc89628f450e8b1e0c6fca4ec3f7b34cccf55f3f531c659ff4d79'
```



```

(fno , inputa , outputa , locktime , trand) = tran(thash , 0)
strd = pybitcointools.deserialize(trand.encode('hex'))
phash = strd['ins'][0]['outpoint']['hash']
outi = strd['ins'][0]['outpoint']['index']
print "Previous Hash %s" % phash
print "Output index %d" % outi
scr = strd['ins'][0]['script']
scrd = scr.decode('hex')
siglen = ord(scrd[0])
siga = scrd[1:siglen+1]
sige = siga.encode('hex')
pkeylen = ord(scrd[siglen + 1])
pkey = scrd[siglen+2: siglen+1+ pkeylen+1]
pub = pkey.encode('hex')
print "Public Key is %s" % pub
print "Signature is %s" % sige

(fno , inputa , outputa , locktime , trand1) = tran(phash , 0 )
oscr = outputa[outi][2]
print "Script Output %s" % oscr
strd['ins'][0]['script'] = oscr
strd1 = pybitcointools.serialize(strd)
ans = pybitcointools.ecdsa_tx_verify(strd1, sige, pub)
print "Validation %r" % ans

```

Output

```

Previous Hash
a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d
Output index 0
Public Key is
042e930f39ba62c6534ee98ed20ca98959d34aa9e057cda01cfd422c6bab3667b7642652
9382c23f42b9b08d7832d4fee1d6b437a8526e59667ce9c4e9dcebcabb
Signature is
30450221009908144ca6539e09512b9295c8a27050d478fbb96f8addbc3d075544dc4132
8702201aa528be2b907d316d2da068dd9eb1e23243d97e444d59290d2fddf25269ee0e0
1
Script Output 76a91446af3fb481837fadbb421727f9959c2d32a3682988ac
Validation True

```

Finally, we have reached the Holy Grail, we have verified a real Bitcoin transaction.

Once again, let's start from the beginning. The owner of the Bitcoin transaction with a hash value starting with cca75 signs the transaction and leaves the signature in the input part of it. To confirm whether this signature was signed with a certain private key or not, a public key is required. It is with the private key, that we get access to entities like the public key and the Bitcoin address (aka RIPEMD hash).

Signing a transaction is executing some code on some transaction data with a private key. The net result of this task is the signature. The owner leaves behind this signature only in the inputs. He/she also adds the public key, that is derived from the private key into this input field.

So, with the same transaction bytes and the signature and public key available to us, we call the function, `ecdsa_tx_verify` and not `ecdsa_verify`. The function verifies if the signature was signed by the same private key, the one that created the public key.

But here, one basic point is missed out and it is the signature of the data comprising a transaction. This signature is stored within the transaction and it becomes an integral part of the transaction data. It becomes a recursive process.

We can however, work on the assumption that the inputs script field is blank and then calculate the signature on this transaction data. Then, this signature and public key can be added to the inputs script field. But, it will change the transaction data again and a new signature has to be calculated. This process will go on for ever and ever.

The next best alternative is to zero out the scriptsig field. Then calculate a new the signature. This signature does not need any verification or recalculation.

The brilliant man, Mr. Satoshi did something different. He replaced the scriptsig field in the input with the scriptpublickey of the output of the transaction hash found in the input. Now, there is no choice now but to go to the output referenced by the transaction hash. From this output, the scriptpublickey is retrieved using the output index, which is stored in the variable `outi`.

The same tran function is executed again and the list member denoted by variable `outi` is accessed. The blk file number is assumed to be 0.

The scriptpublickey is saved in the variable `oscr`. This variable also contains some script opcodes. The important thing is the 20 byte RipeMD hash, which eventually gives the Bitcoin address.

The data, on which the signature will be computed, also contains the rules for claiming ownership. The scriptpublickey field gives this information as well.

In our code, the only scriptsig member is replaced with the recently fetched scriptpublickey, this value is in variable `oscr`.

Now, we have a series of hex bytes with some of the data changed. To achieve the original bytes in string form, the `serialize` function is used on the `strd` variable. Variable `strd` is a python dictionary, it cannot be used directly. And hence, the string variable `strd1` is used.

The `verify` function requires the same three parameters, the transaction data in variable `strd1`, followed by the signature from the inputs and the public key also from the inputs. If all goes well, the answer will be true. And it is.

A miner verifies the signature of a transaction in this manner. Anyone can confirm that the owner of this public key signed this transaction. If you comment out the line where we replace the script field, a value of false will be returned.

Checking the Signature for a Transaction with Using Only the Inputs

```
ch1005.py
import pybitcointools
from cfuncs1d import *
thash = '756c1cf676c73b951ecb3b281b375858938b503c2b9b296d9d1cd59e839daea0'
(fno , inputa , outputa , locktime , trand) = tran(thash , 132)
strd = pybitcointools.deserialize(trand.encode('hex'))
phash = strd['ins'][0]['outpoint']['hash']
outi = strd['ins'][0]['outpoint']['index']
print "Previous Hash %s" % phash
print "Output index %d" % outi
scr = strd['ins'][0]['script']
```

```

scrd = scr.decode('hex')
siglen = ord(scrd[0])
siga = scrd[1:siglen+1]
sige = siga.encode('hex')
pkeylen = ord(scrd[siglen + 1])
pkey = scrd[siglen+2: siglen+1+ pkeylen+1]
pub = pkey.encode('hex')
print "Public Key is %s" % pub
print "Signature is %s" % sige

(fno , inputa , outputa , locktime , trand1) = tran(phash , 132)
oscr = outputa[outi][2]
print "Script Output %s" % oscr
print "Length of ins is %d" % len(strd['ins'])
strd['ins'][0]['script'] = oscr
strd['ins'][1]['script'] = ''
strd1 = pybitcointools.serialize(strd)
ans = pybitcointools.ecdsa_tx_verify(strd1, sige, pub)
print "Validation %r" % ans

```

Output

```

Previous Hash
6386c136bc8c1e6b8ae7b51d6444ccfc87dfcb0b334b4cb2ca29bad08c4fb00c
Output index 1
Public Key is
04532f5d9b05b4f826e3e1576a48db348bb9d25d0ebdaae3a2ce704793cb82b37e724a6
37b4b3501420a8d65325d645b72a35e6697e0ab7accf765a3c5a7a1015a
Signature is
3044022047f4f9b930fa2f9161300fdc6b4529b9b5fd0bbc0eb1acc646b1ca4633bfad850
220553ecc57e6e8b9c1c5c368100bf8baa5fd260573a2ba7536ea46247ddacdc33101
Script Output 76a914331e4ec2b58e17a406643659d2c8aa70449c226e88ac
Length of ins is 2
Validation True

```

Most transactions have more than one input, the previous transaction had multiple outputs but only one input. Block number 129 has one such transaction,

```
756c1cf676c73b951ecb3b281b375858938b503c2b9b296d9d1cd59e839daea0
```

with two inputs.

The block number 132 has multiple inputs, at least 2, which point to different transaction blocks on disk. It takes time to fetch the block. So, we play safe. If you hate waiting, choose block number 129 instead. Since we are moving backwards, larger the number the more time the program will take, but at least the output will be similar. If things do not work, use a slightly larger block number.

The Satoshi rule is that for multiple inputs, their scriptsig members are set to a null string. The rule is followed and ['ins'][1] is set to a null string. The signature will not get verified if this step is avoided. We are assuming there are only 2 inputs in the transaction, which may not be the case.

The length of the ins field is 2 as the list has 2 members.

Validating a transaction with multiple inputs

```
ch1006.py
import pybitcointools
from cfuncs1d import *
thash = '756c1cf676c73b951ecb3b281b375858938b503c2b9b296d9d1cd59e839daea0'
(fno , inputa , outputa , locktime , trand) = tran(thash , 130)
strd = pybitcointools.deserialize(trand.encode('hex'))
phash = strd['ins'][0]['outpoint']['hash']
outi = strd['ins'][0]['outpoint']['index']
scr = strd['ins'][0]['script']
scrd = scr.decode('hex')
siglen = ord(scrd[0])
siga = scrd[1:siglen+1]
sige = siga.encode('hex')
pkeylen = ord(scrd[siglen + 1])
pkey = scrd[siglen+2: siglen+1+ pkeylen+1]
pub = pkey.encode('hex')
(fno , inputa , outputa , locktime , trand1) = tran(phash , 130)
oscr = outputa[outi][2]
strd['ins'][0]['script'] = oscr
for i in range(1 , len(strd['ins'])):
    strd['ins'][i]['script'] = ""
strd1 = pybitcointools.serialize(strd)
ans = pybitcointools.ecdsa_tx_verify(strd1, sige, pub)
print "Validation %r" % ans
```

Output

Validation True

In the last program, we assumed that there were only 2 inputs. This last program simply automates the process of setting the input field to null since there can be varying number of inputs. All of them are set to "" in a for loop.

The for loop starts from 1 as the script of the first input is taken care of. The counting starts from 0 and the loop ends at one less than the list size.

As we understand other types of signatures in the future course, we will scan through every Bitcoin transaction and check if the signature was verified correctly. It is a very slow process though.

To summarize, given a transaction hash and no access to the private key, we can verify if the signature is valid. The script field of the inputs must be replaced by the script public key of the output, which is referenced by the transaction hash in the input. All the other inputs script fields must be set to a "" string.

CHAPTER 11

Roll your own transaction

This chapter is only the start of the Holy Grail of Bitcoin. Here, we attempt to create a Bitcoin transaction manually, byte by byte, well almost close enough to the bare metal. By the time, we come closer to the last few chapters of this book, all code will be written by us. Though it is a long arduous journey which almost zapped our thinking processes, please do not loose faith. This line is our constant refrain.

Two programs are to be downloaded, as they do not come with the Bitcoin software. Both are command line programs. The first program is bitcoind, which is a server and the second program is a client called bitcoin-cli.

The curl program is used to download these utilities.

```
curl -O https://bitcoin.org/bin/bitcoin-core-0.13.1/bitcoin-0.13.1-osx64.tar.gz
or
curl -O https://bitcoin.org/bin/bitcoin-core-0.12.1/bitcoin-0.12.1-osx64.tar.gz
```

Or better still use the following URL to decide which version, <https://bitcoin.org/bin/>

We initially started with version 0.12 and then upgraded to 0.13/0.14/0.15 upon its release. It does not matter as to which of these you use. Hopefully, by the time our book is released, the miners would have approved the 0.16 version.

Our journey of learning Bitcoins started with version 0.11. Then we experimented on multiple versions of the python bitcoin libraries. Along the way, we tweaked the Bitcoin source code multiple times and therefore rebuilt it, many a times. At times things work but multiple times, they just crashed. But, hang in there..

The tar command with the following parameters untars the downloaded file.

```
tar -zxf bitcoin-0.12.1-osx64.tar.gz
or
tar -zxf bitcoin-0.13.1-osx64.tar.gz
```

The tar command creates a folder called bitcoin-0.13.1 with three sub folders. The most important one is the bin folder which contains five executable files. We will use only two of them. The include and lib folders can be ignored for the time being.

First, create a folder called /usr/local/bin with the following command, if not present.

```
sudo mkdir -p /usr/local/bin
```

The -p option is for quiet mode, so no error messages are displayed on the screen. The command works only for the Mac and Linux, the rules for the Windows OS change. The Mac OS X is more Linux than Windows. The sudo command gives root access to the user, so no rules apply.

We copy these executables to the bin folder.

```
sudo cp bitcoin-0.13.1/bin/bitcoin* /usr/local/bin/
```

In the default folder /Users/vijaymukhi/Library/Application Support/Bitcoin, there is a file called bitcoin.conf.

The contents of the file bitcoin.conf are

```
bitcoin.conf
rpcuser=bitcoinrpc
rpcpassword=mukhi
```

If this file is not present, you can create one yourself. The Bitcoin help has a convoluted set of commands that you can simply copy and paste in your terminal. This file must be in the folder /Users/vijaymukhi/Library/Application Support/Bitcoin. The next best alternative is to use the text editor and create this file with the above lines in the default folder.

The password here is my family name, mukhi; you can choose your password. This file must in the default folder. The default folder in other operating systems will be different and the commands will change accordingly.

In the next chapter, we will demonstrate the significance of this file as it reflects Bitcoin's mood and behavior.

Once the environment is set, we start creating our first transaction which transfers Bitcoins to some other Bitcoin address. We first need a Bitcoin address with some Bitcoins. These Bitcoins will be transferred to a new owner. Unless a Bitcoin address does not have/own any Bitcoins, it cannot transfer them to another address.

Bitcoin prices are growing though the roof. There is an Indian company called Zebpay from where we buy our Bitcoins. We also have a Bitcoin wallet at another company called Unocoin and the US giant blockchain. Bitcoin wallets on the Android and Apple's iPhone may be the best thing since sliced bread. Unfortunately, they are too high level and constrained. They do not allow any access to one's own private key. No private key, no signing of a signature. Also, code written in Python does not run on phones at all.

A repetition, a private key is used to compute the public key. It is twice over hashed and then base58 encoded. Finally, a checksum is computed. This process is only one way and given a Bitcoin address, it is impossible to compute any other previous artifact like the RipeMD hash value etc. The private key is used to generate the Bitcoin address when Bitcoins are to be transferred to any other address. The wallet has our private key and hence it can transfer Bitcoins on our behalf. The private key is the only proof of ownership of Bitcoins.

The desktop Bitcoin wallets can create Bitcoin addresses in the millions. They also give access to the private key. There are a million desktop wallets including Bitcoin-Qt but we choose our humble bitcoin-cli, which is low level stuff, non gui.

First, let's start the Bitcoin server with the following command

```
$bitcoind -daemon
```

Output

```
Bitcoin server starting
```

```
Error: Cannot obtain a lock on data directory
```

```
/Users/vijaymukhi/Library/Application Support/Bitcoin. Bitcoin Core is probably already running
```

The error comes in because our GUI Bitcoin core is running on the computer. The GUI and the command line server cannot run in parallel as they use the same port number, technically. A better explanation is that they both share the same codebase. The GUI server must be shut for the bitcoind program to run.

The -daemon option is part of Unix folklore. Please note, it is not a misspelling, it is not demon as in angels and demons. Without the -daemon option, the command window will be of no use to anyone. Servers must always run in the background as they need no user input.

```
$bitcoind -daemon
```

Output

```
Bitcoin server starting
```

The server takes about 10 seconds to warm up before attending to our series of commands.

```
$bitcoin-cli getblockhash 1
```

Output

```
00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048
```

The client program, bitcoin-cli understands about a 100+ commands, one of them being getblockhash. This option simply returns a hash value of the given block number, in our case block number 1. Please feel free to check the above displayed hash value of block 1 in blockchain.info.

The client in all client server interactions does very little work. Servers are always more intelligent than clients. This rule applies to the Bitcoin universe as well. The client simply passes the two parameters, getblockhash and 1 to the server using networking technologies. The server obtains the calculated block hash value and then returns the data to the client who dutifully displays it.

There are two chapters in the book on Bitcoin networking. And, a little later in this chapter, we have written python programs mirroring the bitcoin-cli program.

```
$bitcoin-cli getaddressesbyaccount ''
```

Output

```
[
  "1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n",
]
```

The getaddressesbyaccount method asks the server to enumerate all the past Bitcoin addresses in a bitcoin wallet. These addresses are stored in a file on disk, aptly called wallet.dat.

Your output will be different, you will not see our Bitcoin addresses. So, the Bitcoin addresses we use in our code must be replaced with the values you see. Make changes to your code before you run them, we will inform you accordingly most of the time.

The next command creates a new Bitcoin address on the fly.

```
$bitcoin-cli getnewaddress ''
```

Output

```
1DTGvGaFHfeHvbZThZrPbddM4KgF6E4QHn
```

The method getnewaddress creates a new Bitcoin address. Now let's check if this address is added to the wallet.

```
$bitcoin-cli getaddressesbyaccount ''
```

Output

```
[
  "1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n",
  "1DTGvGaFHfeHvbZThZrPbddM4KgF6E4QHn"
]
```

There are two Bitcoin addresses in our Bitcoin wallet.

The address beginning with 1Lg5 is used throughout this chapter, please replace it with a Bitcoin address created using your private key. You cannot simply copy cut paste the code in this chapter and assume that it will work, but you can verify our work, which you must.

The bitcoin-cli is not written in Python or C, but in C++. It uses a lot of arcane features of C++. The Bitcoin client bitcoin-cli however incorporates the python list concept, wherein multiple values are returned in a list []. It uses json also.

So far, the Bitcoin address, which is the source for our Bitcoins is obtained. The next task is to acquire the private key that generates this address. And, it is achieved by executing just one more command.

```
$bitcoin-cli dumpprivkey 1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n
```

Output

```
L2T9wdbjAvyiXQnm27WPNR323Ce8cDkDkMARQx125CQZ43gHG4yR
```

The method dumpprivkey takes a Bitcoin address present in the wallet and returns a private key. The purists will be antagonized as the private key is not to be published and they are right. However, with the private key there is control on the ownership of the Bitcoins, the only catch here is that currently there are effectively zero Bitcoins owned by this Bitcoin address. Please verify this using your blockchain explorer.

The private key beginning with L2T must be replaced with your private key throughout the chapter.

To prove our mettle, we logged in to Zebpay, our Indian Bitcoin wallet and transferred a minimum of Rs. 15 or more worth of Bitcoins to this Bitcoin address. This is the minimum value we can transfer at one time with our mobile Bitcoin wallet app. We transferred this small quantity of Bitcoins about 400 or 500 times during the writing of this book, so we have multiple unspent Bitcoin outputs during the last two years. The Zebpay app created for us the following Bitcoin address which we on paper own: 1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv. The point to be noted here is that Zebpay does not give away the private key to us, but Bitcoins can be safely transferred to this address.

We request you to transfer Bitcoins to your address not ours, though it can be transferred to this address. We will not complain.

Displaying Unspent Outputs of a Bitcoin Address

```
ch1101.py
import bitcoin
addr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
unspend = pybitcointools.history(addr)
print unspend[0].keys()
for obj in unspend:
    print obj
```

Output

```
Fetching more transactions... 50
['output', 'block_height', 'spend', 'value', 'address']
{'output':
u'2798f73adddd8d4b407ed6153045bdf6c5d4d72b6d1454d20456cd5eece4447e:3',
'block_height': 446079, 'spend':
u'498a8a4ca188c655a9c0d43e6cb43d99501d2874a190feb926e37d3255d2d8d7:0',
'value': 58900, 'address': u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}
```


Critical observation: The pybitcointools library worked as advertised in the year 2016. Then things stopped working, so we moved to the bitcoin library. Please install this bitcoin library as

```
$sudo pip install bitcoin
```

The code on disk now uses this newer library. Also, change the variable name bitcoin to bitcoins in the code whenever it occurs.

One approach to unravelling the unspent Bitcoins is to write this Bitcoin address in blockchain.info. It will display a list of unspent outputs. However, what is most important here is the net available balance of Bitcoins in this bitcoin address.

The wallet.dat file stores the transactions that transferred these bitcoin to and fro from our Bitcoin address. To determine if a Bitcoin output is unspent, there should be no reference made by any input of a transaction to this output. However, the Bitcoin world does not keep any running balance of the outputs of every Bitcoin address, unlike the other blockchains.

The input points to an output and only outputs own Bitcoin addresses. Inputs refer to these addresses indirectly, through a transaction hash value and an index offset.

An output can be referenced only once by an input transaction hash value. There can also be no part transfer. If all the bitcoins in the output are not consumed, the miner takes the rest of it. The miner's fee is sum of all inputs - sum of all outputs.

Thus, simply looking at the outputs, one cannot discern if the bitcoins in the output have been spent or not. There is no field to determine the same. The reason being, no-one can change a byte of any past/previous block. When an owner receives bitcoins, it is basically the bitcoins unspent by the previous owner. And, when the owner transfers these Bitcoin to the next owner, the bitcoins are deemed to be spent. But there is no field or indicator for it. The next best option is to move back in time in the blockchain and make a change to a certain field. This is simply not possible because the blockchain cannot be changed.

So, we come back to the same question as in how do we determine that a Bitcoin output is spent or not?

This is a very slow and tedious process. It starts with an output Bitcoin address. The inputs of all the transactions till the latest transaction is scanned forwards and not backwards. A check is performed only on those transactions inputs that reference two fields. These are, the transaction hash value and the output index number. Both these fields are stored in the inputs, one after the other.

The reason behind scanning forwards and not backwards is that unless Bitcoins are not received in the past, they cannot be transferred to someone else. The cart is never placed before the horse. If an output is not referenced by any previous input, then it is a proven fact that this output is not spent.

It would take a lifetime to write code for this task. So, we use a function called history in the bitcoin module that keeps track of all spent and unspent outputs. The history command returns a list of dictionaries. The list has a history of all transactions of a certain Bitcoin address. The list variable unspent returned by the history function discloses all the transactions that have taken place with a Bitcoin address.

Our Bitcoin address has been extensively used. Your numbers will change and with time, the history will keep growing.

Let's take the 0th item in the list. The result on screen looked like the following {'output': u'2798f73adddd8d4b407ed'. The key output is a transaction hash value starting with 2798. This is the transaction hash value of the address that supplies the Bitcoins.

Enter this transaction hash value in blockchain.info and look at the fourth Bitcoin address on the right which stands for outputs. There is 1Lg5, our bitcoin address. This value appears on the top right of the webpage. The transaction hash value has a colon and then a 3 following it. It indicates that this Bitcoin address is fourth on the list. This number 3 is the output index position. The year 2017 is when this transaction was created.

Coming back to the other fields returned by the history function, there is `block_height` which is the 446079, it matches with information displayed in our blockchain explorer, `blockchain.info`. The Bitcoin value 58900 BTC is also seen in the explorer. Only the display part varies. The history function prefers 589000. Both have the same address as well.

Coming back to the code, each list is displayed in the for statement. For space reasons, a truncated version is displayed on the screen.

There is a key called `spend` to confirm that the output has been spent, but the key is not used. Please check this transaction hash in the explorer.

Every time a Bitcoin transaction takes place, a database is updated and a history is created. So, to say that bitcoin stores the data on every Bitcoin address is not true. The history function navigates to one of the many websites, to which it requests, for a history of all transactions related to a certain Bitcoin address. These websites do not scan through each transaction, they simply access the history stored in a database on their computers.

The Bitcoin wallet sitting on our hard disk performs the same task but only for the Bitcoin addresses created in the wallet, it is at a smaller scale.

A clarification. The later code may not run on your machine when you read this chapter, as the Bitcoin address 1Lg5 will have no Bitcoins left by then. However, you can first transfer some Bitcoins to address 1Lg5 and then try. Our code will work like a charm. It must be noted that the private key of this Bitcoin address is public, so anyone can claim ownership of these Bitcoins and they will.

The history shows multiple transactions with this address but the fun is in seeing it with transactions you create yourself.

Finding a Transaction having an Output with Unspent Bitcoins

```
ch1102.py
import pybitcointools
addr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
unspend = pybitcointools.history(addr)
unspendl = []
i = 0
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)

spendd = unspendl[0]
print spendd
newobject = spendd['output']
ind = newobject.index(':')
thash = newobject[0:ind]
outi = int(newobject[ind + 1:])
print thash
print outi
print len(unspendl)
```

Output

```
{'output':
u'7669ad0f5cf51653258a69addc50456d372fa62c22c8c3ab301fdd25eff1e92e:1',
'block_height': 452716, 'value': 66600, 'address':
```

```
u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}
7669ad0f5cf51653258a69addc50456d372fa62c22c8c3ab301fdd25eff1e92e
1
5
```

There is one change made to the last program. As before, the unspent list returned by the history function is scanned. A check is performed for a key called `spend` in the list. If such a key exists, it indicates that this output has already been spent or its ownership has been transferred. This means that some input somewhere has this transaction hash value and an output index that refers to this output.

The keys displayed are `output`, `block_height`, `value` and `address`.

As the history function returns `spend` and `unspent` outputs, so a subset list called `unspendl` is created to hold those outputs that do not carry the `spend` key. The `if` statement acts like a filter. The topmost member of the list of unspent transactions is shown.

Having a list of transactions whose outputs are already spent is useless as the ownership of these Bitcoins is already transferred and it belongs to the new owners. The miners also reject these transactions. In Bitcoin jargon, it is called double spending. If the Bitcoin world allowed double spending, then its value as a crypto currency would drop down to zero. There is a lot of effort that goes into tracking which Bitcoins are not spent.

Using the variable `spendd`, the transaction hash value and output index will point to the output bringing in the Bitcoins. You can confirm it in a blockchain explorer. These members are part of the value pair of the output key. The two values are separated by a `:`. The `index` function of the `string` module gives the start position of colon: and the `slice` operator gives the transaction hash. A simpler approach would be to hard code a value in the slice operator, as the transaction hashes are always 256 bits large or 32 bytes large.

A colon is where the transaction hash value ends and the output index starts. The output index can be a large number as there can be multiple outputs in a transaction. Everything from the `:` onwards to the end of the string is read. This starting point of the colon is stored in the `ind` variable. The string is converted into a number using the `int` function. This step is important as the output index may not be a single digit.

Honestly, we have never got a right answer with the slice operator in the first attempt, we are always off by one, one too many or one too less. We have five unspent transactions with unused Bitcoins, but not for too long.

We insist that you key in the transaction hash value starting with 7669 in the blockchain explorer. The outputs displayed on the right of the arrow shows the second output having the Bitcoin address starting with 1Lg. So far, this output is not spent, so `unspent` is tagged in the output. Thou will see spent.

We earlier used the `tran` function to indirectly return the `scriptpublickey` of the output. But it takes too long, so let's find a better way to get details of a transaction.

For the record, the history command also uses the `blockchain.info` web service to fetch the history. Please make sure that you are connected to the internet. At times, web services will fail, so keep trying.

We first install one more python module called `blockchain`.

```
$sudo pip install blockchain
```

Reading the transaction data from an external webstore

```
ch1103.py
from blockchain import blockexplorer
tx =
```

```
blockexplorer.get_tx('cb67e6f9e120bb1fad3ef99b479aabc735a797ac5e59ae8829e15
49749c4d84b')
print tx.block_height
print tx.hash
print tx.size
out = tx.outputs
print len(out)
print type(tx) , type(out)
for o in out:
    print "n" , o.n
    print "value" , o.value
    print "address" , o.address
    print "tx_index " , o.tx_index
    print "script" , o.script
    print "o.spent " , o.spent
    print
```

Output

```
417652
cb67e6f9e120bb1fad3ef99b479aabc735a797ac5e59ae8829e1549749c4d84b
436
4
<type 'instance'> <type 'list'>
n 0
value 57700
address 18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7
tx_index 156940719
script 76a91457b138abaa1b81a766f18b79b6b2d605ea58dee188ac
o.spent True
n 1
value 68910000
address 381C2MCxaM83upgFpbgp4g7ezeveApEzK
tx_index 156940719
script a9144540ac53786b5c6b77b1b8823896b009e893fb2187
o.spent True
n 2
value 41290000
address 38DKJ3kx2BuF6JCBKqEcnaazCwsUkhWF8X
tx_index 156940719
script a914478bbbe3c42419a9ad6c0e52cb41ed25c267f0af87
o.spent True
n 3
value 107543611
address 3NbRDVgga3iTGKtRdsUVN4cvFvV6Cfwy8E
tx_index 156940719
```

```
script a914e54b96cefb649b6cc53af91aefc84c474c4dc6f87
o.spent True
```

There are many python blockchain libraries. The function `get_tx` from the `blockchain` module takes a transaction hash value. It then returns a transaction object that has all the details of the transaction, like the inputs and the outputs. Some of these fields may have only one value but there will be many values for the inputs and the outputs. The program displays members like the block number the transaction resides in, it's size in bytes etc. The variable `out` is a list. The outputs in the transaction object `tx` are accessed using the field called `outputs`, which represents the outputs in the transaction. A for loop iterates through them.

The field `scriptpublickey` which is called `script` is of the essence here. This function and the `history` function can only work if all the Bitcoin data is stored in a database with different data structures.

The `pybitcointools` module uses the same online database that `blockchain` uses. In fact, `pybitcointools` have proclaimed that it uses the same blockchain api.

The advantage of using the `blockchain` api over our written code is that it internally scans all the physical block files and it is much faster and cleaner.

Adding an Extra Function to Return the Scriptpublickey of the Output

```
cfuncs1d.py
from blockchain import blockexplorer
def scrpubkey(thash , outi):
    tx = blockexplorer.get_tx(thash )
    out = tx.outputs
    length = len(out)
    for i in range(0, length ):
        if i == outi:
            return out[i].script
```

There is a function called `scrpubkey` added at the end of the file `cfuncs1d.py`. This function simply borrows code from the last program and returns the `scriptpublickey` member of the output. We could have used the `tran` function, but it is very slow.

Retrieving the outputs `scriptpublickey` without reading from disk

```
ch1104.py
import bitcoin
from cfuncs1d import *
addr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
unspend = bitcoin.history(addr)
unspendl = []
i = 0
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)
spendd = unspendl[0]
print spendd
newobject = spendd['output']
```

```
ind = newobject.index(':')
thash = newobject[0:ind]
outi = int(newobject[ind + 1:])
print thash
print outi
print "Starting the faster method"
scr = scrpubkey(thash , outi)
print scr
```

Output

```
{'output': u'7669ad0f5cf51653258a69addc50456d372fa62c22c8c3ab301fdd25eff1e92e:1',
'block_height': 452716, 'value': 66600, 'address':
u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}
7669ad0f5cf51653258a69addc50456d372fa62c22c8c3ab301fdd25eff1e92e
1
Starting the faster method
76a914d7ce0c2c237ec0ec04931335377a87d16b9865ad88ac
```

Like the code in the previous chapter, we need to sign a transaction. In the inputs, there is a field called signature and public key. However, the irony here is that the signature value is not available as we are in the process of computing a valid signature. So, as before, some transaction gives us the Bitcoins to be transferred. From this output, the scriptpublickey field is obtained. This value is then placed as the value of the input signature+public key field. Also, all the other input rows, if any, are nulled out. All the outputs are also ignored. The ground work is now set for the data to be signed.

The transaction hash value and the output index are obtained as before, using the first list member from the history function. Then, the newly minted function `scribpubkey` is used to return the `scriptpublickey`. This method is much faster than the `tran` function. The only downside with these api's is that if the server is down or your Internet is not working, you will have no choice but to sleep over it.

Creating a Rudimentary Bitcoin Transaction by Hand

[illegible]

Output

Size 85

Hash bytes

30409ff32a1ad88ac00000000

175

```
    ]
  }
}
]
```

It is now time to create and sign our own transaction. This transaction will have only one input and one output. Later, we will sign this transaction without using a library function.

Finally, we will send this signed transaction to miners all over the world so that they can include our transaction in a block which the whole world will see.

So, let's once again go through the different fields of a transaction. It is a transaction and not a block. The miners put the transactions together in a block. The magic bytes, the block size as well as the block header are once again, the concern of miners. Most of the fields in the transaction are hard coded.

The first field, 4 bytes is the transaction version and the variable used is `tver`. The version field has a value of 1. The next field is the number of inputs, for some reason everyone calls inputs as `vin`, so have we. In our case, `vin` is a variable that determines the number of inputs in our transactions. All these inputs structures are stored back to back.

The first field in the inputs is the transaction hash value of the output bearing gifts or bitcoins. This hash value is always displayed in a unique way. Every two characters are reversed and then the last byte is bought as the first and so on and so forth. Variable name used is `dhash`.

Then are the 4 bytes for the output index in the above transaction id. Since it is the first output, therefore it has a value of 1. The variable name is `outi`. The order of the bytes storing the number 1 is significant.

The length of the `scriptsig` is 0 as the signature is not yet computed. Nevertheless, the public key is available. The variable `scri` uses two 0's as it is a string. Finally, the inputs end with the sequence number which is always `f's`. The variable name is `seqno`.

Next comes the outputs which are much simpler. The variable is called `vout`. This variable represents the number of outputs, here there is only 1 output.

This is followed by the bitcoin value that must be converted from a simple number to a value which occupies 8 bytes of space. `0a` in hex is 10 in decimal. A point to be noted here is that we were forced to use the `pack` function of the module `struct` with the `Q` modifier.

The length of the public key is 25 bytes as this is the only type of transaction for now and the opcodes remain the same till time immemorial. Only the `RipeMD` hash value will change depending upon the Bitcoin address.

The initial two opcodes and the final two opcodes bytes are the same. The length of the hash will be `0x14` or 20 bytes long. Only the hash value changes. The hash is the `RipeMD` hash value of the public key of the new owner. It is our Zebpay Bitcoin address `1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv`. In a sense, we are sending money to ourselves and not to anyone else.

The `locktime` variable is all 0's. All the above variables are concatenated in the right order in a string called `tx`. The decoded size is displayed as 85 bytes. The `SHA` hash value of the decoded bytes is also calculated in the `hash` variable and then displayed after reversing them. The hash value starts with `0cd2`.

Thereafter, the `decoderawtransaction` method of the bitcoin client `bitcoin-cli` is used which converts these bytes into a readable transaction. If the bitcoin module is imported, then function `deserialize` can be used. It is not difficult to convert raw transaction bytes into readable transaction data and vice versa.

The newly calculated hash value matches up with the one displayed by `bitcoin-cli` client. The `txid` field of the output also starts with `0cd2`.

The method `decoderawtransaction` has a lot more to offer. It calculates the sha256 hash value of the transactions and it also publishes the size of the transaction, like the `len` function.

The order of display is not in the same as the bytes in the string as the last field, locktime comes first. In the inputs, the `txid` field is reversed therefore a hash value that ends in 3412 but is displayed as 1234. The `scriptSig` field is blank.

The outputs also have many more add-ons. Not only do we see the raw hex bytes of the `scriptPublicKey` but also the assembled bytes. It however, does not display the length of the RipeMD hash value. The rationale is that when you have the actual hash value, there is no need for length.

The bonus here is that the RipeMD hash value is converted to our Zebpay Bitcoin address. The output is enclosed in dictionary like braces. The inputs and outputs are enclosed in a list.

Most of the code to decode the transaction in the program is from the Bitcoin source. This code has been written by the Bitcoin gods themselves.

Filling Up Some Blanks Before Sending Our Transaction to Miners

```
ch1106.py
import bitcoin
import subprocess
import os

saddr = "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
bitcoins = 0.000654
iaddr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
unspend = bitcoin.history(iaddr)
unspendl = []
i = 0
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)

spendd = unspendl[0]
print spendd
newobject = spendd['output']
ind = newobject.index(':')
thash = newobject[0:ind]
outi = int(newobject[ind + 1:])

str0 = [{"txid": "%s", "vout": %d}] % (thash, outi)
str1 = '{ "%s" : %f}' % (saddr, bitcoins)

raw = subprocess.check_output(["bitcoin-cli", "createrawtransaction", str0, str1])
print raw
os.system("bitcoin-cli decoderawtransaction " + raw)
```

Output

```
{'output':
u'7669ad0f5cf51653258a69addc50456d372fa62c22c8c3ab301fdd25eff1e92e:1',
'block_height': 452716, 'value': 66600, 'address':
u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}
01000000012ee9f1ef25dd1f30abc3c8222ca62f376d4550dcad698a255316f55c0fad697
```

```
601000000000ffffffffff0178ff00000000000001976a914c847b6d84ecf8048474b19c4a23304
09ff32a1ad88ac00000000
{
  "txid": "0688896c3dfc2eb41c2190efbd31b765b8386b3d8ac466e4078c00870f7aad5f",
  "hash": "0688896c3dfc2eb41c2190efbd31b765b8386b3d8ac466e4078c00870f7aad5f",
  "size": 85,
  "vsize": 85,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7669ad0f5cf51653258a69addc50456d372fa62c22c8c3ab301fdd25eff1e92e",
      "vout": 1,
      "scriptSig": {
        "asm": "",
        "hex": ""
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00065400,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1KFz4ACsUqeFrJsFD2P3YgU7zS4WEs0otv"
        ]
      }
    }
  ]
}
```

The earlier program had just moved an inch towards sending our raw transaction. We try to move faster now. The above program has most of the data we keyed in the earlier program, like the version number etc. These are standard fields. Then, instead of creating the entire transaction step by step, variable by variable, only the changed data is provided. The client program, bitcoin-cli creates the entire transaction and the method createrawtransaction is used for this purpose.

The system function from the os module is not used, instead the check_output function from the subprocess module is used. Both in a sense, execute command line programs with parameters.

There are two reasons for not using the system command. One, the module subprocess gives access to the return value after the command executes. In this specific case, the createrawtransaction method returns the bytes of the

transaction it has created. The second problem with the system command is that it takes one large string, whereas the `check_output` uses a more complex and nuanced approach since it uses parameters. The `createrawtransaction` method however, requires the inputs and the outputs as separate parameters.

To be precise, the function `check_output` takes only a single parameter, which is a list. It is a list of four members. First is the name of the program to be called, `bitcoin-cli`. Then, the second member of the list is a method, i.e. `createrawtransaction`. Finally, two more parameters which are string values denoting the inputs and then the outputs in lists.

First, a string `str0` is created where a list of dictionaries is specified. The various input fields are specified by their key names. The hash value of the transaction supplying Bitcoins is first given. The key name used here is `txid`. Just a reminder, this transaction hash value is acquired using the `history` function and the `output` key. Next in the `wish` list, is the output index which gives the output index in the transaction hash. The key name is `vout` and its value is stored in variable `outi`.

So far two variables, `thash` and `outi` are used to create a simple dictionary. This dictionary has two keys. The string `str0` is a list with a dictionary. The list concept fits perfectly here as many more inputs can be added, otherwise every transaction would have only one input. So, if this list has 10 members, then there are 10 inputs in our transaction. All this is packaged in just one variable which is then passed as the third list member to the method `check_output`.

The second string `str1` is again a simple dictionary with multiple outputs. Consistency thy name is `bitcoin-cli`.

The rules used for inputs are applied for the outputs also. The dictionary values are enclosed in a list. The bitcoin address of the new owner starts with `1KFz`, our Zebpay address, it is saved in the variable `saddr`. The chosen value of the Bitcoins, `0.000654` represents a very small number as you cannot send 0 bitcoins. This transaction takes place many, many, times in our book, and therefore we choose this small value.

The outputs are represented very differently. It is just one string with values separated by a colon. The Bitcoin address is used as a key and the value is the number of Bitcoins to be sent across. Just a dictionary, no list.

With these four fields of information, the client program, `bitcoin-cli` performs its duty. It returns the raw transaction bytes. This string when given to `decoderawtransaction`, displays the transaction in English.

Once again, the client `bitcoin-cli` is given only 4 values. The transaction hash value, the output index, the value of the Bitcoins to be sent and finally the Bitcoin address. Everything else like `locktime`, `sequence` number, etc. is calculated internally.

The input signature and public key in the inputs must be specified. Otherwise it is not a valid transaction to be sent out on the ether. (a pun on Bitcoin's competitor Ethereum which is facing lots and lots of serious issues)

```
$bitcoin-cli validateaddress 1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv
```

Output

```
{
  "isvalid" : true,
  "address" : "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv",
  "scriptPubKey" : "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
  "ismine" : false
}
```

A Bitcoin address is nothing but a RipeMD hash value which has been converted to a base58 encoded string, enclosing a checksum. The question is can we can reverse this process up to a certain extent ?

The method `validateaddress` decodes the Bitcoin address and gives a valid `scriptPublicKey`. As the Bitcoin address starts from 1, everyone knows the opcodes to add. It is part of the Bitcoin standard.

It is next to impossible to retrieve the SHA hash value or the public key to reclaim the Bitcoin address, so its next to impossible to obtain the private key. Encodings on the other hand, can be reversed, as learnt earlier in the Bitcoin chapter.

To reiterate, if the Bitcoin address is given to the method `validate address`, it gives the original `RipeMD` hash value along with the standard script opcodes. This code gets executed internally by the `createrawtransaction` method. The `scriptpublickey` field is filled up by simply passing the Bitcoin address. Do observe, we are using the hash value 7669 in the inputs.

One Small Step for Mankind, We are Signing the Transaction

```
ch1107.py
import bitcoin
import subprocess
import os
import ast
from cfuncs1d import *
saddr = "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
bitcoins = 0.000654
iaddr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
unspend = bitcoin.history(iaddr)
unspendl = []
i = 0
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)
spendd = unspendl[0]
#print spendd
newobject = spendd['output']
ind = newobject.index('.')
thash = newobject[0:ind]
outi = int(newobject[ind + 1:])
str0 = '[{"txid": "%s", "vout": %d}]' % (thash, outi)
str1 = '{ "%s" : %f}' % (saddr, bitcoins)
raw = subprocess.check_output(["bitcoin-cli", "createrawtransaction", str0, str1])
#print "Raw Bytes to be signed"
print "Last Byte of the bytes returned %d" % ord(raw[-1:])
os.system("bitcoin-cli decoderawtransaction " + raw)

pkey = 'L2T9wdbjAvyiXQnm27WPNR323Ce8cDkDkMARQx125CQZ43gHG4yR'
scr = scrpubkey(thash, outi)
str2 = raw[:-1]
str3 = '[{"txid": "%s", "vout": %d, "scriptPubKey": "%s"}]' % (thash, outi, scr)
str4 = '["%s"]' % pkey
raw1 = subprocess.check_output(["bitcoin-cli", "signrawtransaction", str2, str3, str4])
print "Raw Value"
```

```

print raw1
raw1 = raw1.replace('true', '"true"')
outdict = ast.literal_eval(raw1)
print "After Signing"
print type(outdict)
print outdict
print "complete key=%s" % outdict['complete']
print "Just the field hex"
print outdict['hex']
print
os.system("bitcoin-cli decoderawtransaction " + outdict['hex'])

```

Output

Last Byte of the bytes returned 10

```

{
  "txid": "0688896c3dfc2eb41c2190efbd31b765b8386b3d8ac466e4078c00870f7aad5f",
  "hash": "0688896c3dfc2eb41c2190efbd31b765b8386b3d8ac466e4078c00870f7aad5f",
  "size": 85,
  "vsize": 85,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7669ad0f5cf51653258a69addc50456d372fa62c22c8c3ab301fdd25eff1e92e",
      "vout": 1,
      "scriptSig": {
        "asm": "",
        "hex": ""
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00065400,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
        ]
      }
    }
  ]
}

```

```
]
}
Raw Value
{
  "hex":
    "01000000012ee9f1ef25dd1f30abc3c8222ca62f376d4550dcad698a255316f55c0fad69
    76010000006b483045022100b7ceb2c9be54e3169f6aebc627f9c761b7554899aa78e9e0
    ae0d5446ee44b13202203bddce194059883131d65a1f03e510fc6c4f9588a827787ecbfa
    a4e8f3f8adb30121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd
    365774dce3ffffffff0178ff0000000000001976a914c847b6d84ecf8048474b19c4a233040
    9ff32a1ad88ac00000000",
  "complete": true
}
```

After Signing

<type 'dict'>

```
{'hex':
'01000000012ee9f1ef25dd1f30abc3c8222ca62f376d4550dcad698a255316f55c0fad697
6010000006b483045022100b7ceb2c9be54e3169f6aebc627f9c761b7554899aa78e9e0a
e0d5446ee44b13202203bddce194059883131d65a1f03e510fc6c4f9588a827787ecbfaa
4e8f3f8adb30121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd3
65774dce3ffffffff0178ff0000000000001976a914c847b6d84ecf8048474b19c4a2330409
ff32a1ad88ac00000000', 'complete': 'true'}
```

complete key=true

Just the field hex

```
01000000012ee9f1ef25dd1f30abc3c8222ca62f376d4550dcad698a255316f55c0fad697
6010000006b483045022100b7ceb2c9be54e3169f6aebc627f9c761b7554899aa78e9e0a
e0d5446ee44b13202203bddce194059883131d65a1f03e510fc6c4f9588a827787ecbfaa
4e8f3f8adb30121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd3
65774dce3ffffffff0178ff0000000000001976a914c847b6d84ecf8048474b19c4a2330409
ff32a1ad88ac00000000
```

```
{
  "txid": "12ca43fba825166baba1cb0e376e92fdd3f2ec7c9de83c111d95c7fcc10e6a9c",
  "hash": "12ca43fba825166baba1cb0e376e92fdd3f2ec7c9de83c111d95c7fcc10e6a9c",
  "size": 192,
  "vsize": 192,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7669ad0f5cf51653258a69addc50456d372fa62c22c8c3ab301fdd25eff1e92e",
      "vout": 1,
      "scriptSig": {
        "asm":
          "3045022100b7ceb2c9be54e3169f6aebc627f9c761b7554899aa78e9e0ae0d5446ee44b
          13202203bddce194059883131d65a1f03e510fc6c4f9588a827787ecbfaa4e8f3f8adb3[ALL]
          038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3",
```

```

        "hex":
        "483045022100b7ceb2c9be54e3169f6aebc627f9c761b7554899aa78e9e0ae0d5446ee4
        4b13202203bddce194059883131d65a1f03e510fc6c4f9588a827787ecbfaa4e8f3f8adb3
        0121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3"
    },
    "sequence": 4294967295
  }
],
"vout": [
  {
    "value": 0.00065400,
    "n": 0,
    "scriptPubKey": {
      "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
      OP_EQUALVERIFY OP_CHECKSIG",
      "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
      "reqSigs": 1,
      "type": "pubkeyhash",
      "addresses": [
        "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
      ]
    }
  }
]
}

```

Let's now actually sign this transaction. To accomplish this, the method `signrawtransaction` is called with some more parameters.

All code remains the same till the initialization of the variable `pkey` to the private key. We are currently using the Bitcoin address generated by this private key.

In the past, to verify a transaction, the input signature and public key was set to the value of the `scriptpublickey`. The transaction hash value in the input, along with output index uniquely identifies the script public key or the output. Therefore, the script public key is needed.

Here, the same variable `scr` is used to give the script public key, using the newly added function `scribpubkey`.

Now to signing the transaction.

First, the raw data bytes of the transaction are needed. These bytes are stored in the variable `raw`. These are generated by the method `createrawtransaction`. The lacuna here is that these bytes end in an enter or a number 10. So, slice it. The variable `str2` has the raw bytes returned by the option `createrawtransaction` minus the enter. This variable `str2` is then given as the third member of the list to the function `check_output`. The fourth member of this list is the inputs. The list can have many more dictionary keys. The previous program had only two. If all the keys present in the dictionary are not overridden with values, `bitcoin-cli` takes default values for those missing keys. Therefore, keys like `locktime` and `sequence number` have default values.

The variable `str3` which represents the inputs has an extra key called `scriptPubKey` which is initialized to the `scr` variable. These are simply rules to be followed to the T.

The last list member is the private key used to sign the transaction. This must be the same private key that created the source Bitcoin address. In our case, it is hard-coded in the variable pkey. Change a single byte of the private key and the signature signing process throws an error.

A list is imperative here because in the future, there will be multiple private keys to sign a transaction. The last member of the list should be the words All, we will come back to this value later, as it is optional.

A huge sigh of relief as the output displays the complete key with a value of true. The key hex gives a new set of bytes to work with. These bytes are the same transaction bytes returned by the createrawtransaction but now, with the inputs signature-public key filled.

These bytes are in a string which looks like a dictionary. The unfortunate part is not that the value of the key called complete is true. This value is not in inverted commas so it is not in a string value. To resolve this issue, the replace function in string is used to replace the bool true to a string 'true'. The true is now in inverted commas. This code writing style is called a kludge.

The complete key is the final indicator. If the value of this key is true, then the signature has passed all the verification test.

The module ast has a function literal_eval that converts a string that looks like a dictionary into an actual Python dictionary. This dictionary outdict is used to extract the values of the two keys, hex and complete.

These transaction bytes are run through the decoderawtransaction method where the inputs signature and public key are placed within the key called scriptSig.

Now that the transaction is successfully signed, the next task is to send these raw bytes over. These bytes are present in the dictionary called outdict, the key named hex, i.e. outdict['hex']. In the near future, the hex key will be sent to miners all over the world to confirm this transaction.

Obviously, the transaction hash value or id will also change, as shown by the two outputs. Earlier, the transaction hash value started with 0688 and now it is 12ca.

So, the steps to follow are, first check the input of the bytes returned by createrawtransaction and then check the bytes returned by the signrawtransaction method.

The Last Output has the Inputs Signature and Public Key Filled Up.

```
ch1108.py
import bitcoin
import subprocess
import ast
from cfuncs1d import *
saddr = "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
bitcoins = 0.000154
iaddr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
unspend = bitcoin.history(iaddr)
unspendl = []
i = 0
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)
print "Number of unspent outputs %d" % len(unspendl)
spendd = unspendl[0]
```



```

newobject = spendd['output']
ind = newobject.index(':')
thash = newobject[0:ind]
outi = int(newobject[ind + 1:])
str0 = '{ "txid": "%s", "vout": %d }' % (thash, outi)
str1 = '{ "s": %f }' % (saddr, bitcoins)
raw = subprocess.check_output(["bitcoin-cli", "createrawtransaction", str0, str1])
pkey = 'L2T9wdbjAvyiXQnm27WPNR323Ce8cDkDkMARQx125CQZ43gHG4yR'
scr = scrpubkey(thash, outi)
print "Script public key %s" % scr
str2 = raw[:-1]
str3 = '{ "txid": "%s", "vout": %d, "scriptPubKey": "%s" }' % (thash, outi, scr)
str4 = '{ "s": %f }' % pkey
raw1 = subprocess.check_output(["bitcoin-cli", "signrawtransaction", str2, str3, str4])
raw1 = raw1.replace('true', '"true"')
outdict = ast.literal_eval(raw1)
strd = bitcoin.deserialize(outdict['hex'])
raw2 = subprocess.check_output(["bitcoin-cli", "decoderawtransaction", outdict['hex']])
print raw2
outdict2 = ast.literal_eval(raw2)
print "Transaction ID %s" % outdict2['txid']
scr = strd['ins'][0]['script']
scrd = scr.decode('hex')
siglen = ord(scrd[0])
siga = scrd[1:siglen+1]
sige = siga.encode('hex')
pkeylen = ord(scrd[siglen + 1])
pkey = scrd[siglen+2: siglen+1+ pkeylen+1]
pub = pkey.encode('hex')
print "Signature and Public Key are %s" % strd['ins'][0]['script']
print "Signature is %s" % sige
print "Public Key is %s" % pub
print "complete key=%s" % outdict['complete']
rawoutput = outdict['hex']
raw2 = subprocess.check_output(["bitcoin-cli", "sendrawtransaction", rawoutput])
print "Transaction ID is %s" % raw2

```

Output

Number of unspent outputs 5

Script public key 76a914d7ce0c2c237ec0ec04931335377a87d16b9865ad88ac

```

{
    "txid": "784d182c2f9421654d8e680593cc3be5582ad925a47a0bcaf15227397e045de2",
    "hash": "784d182c2f9421654d8e680593cc3be5582ad925a47a0bcaf15227397e045de2",
    "size": 192,
    "vsize": 192,
    "version": 1,
    "locktime": 0,

```

```
    "vin": [
      {
        "txid": "7669ad0f5cf51653258a69addc50456d372fa62c22c8c3ab301fdd25eff1e92e",
        "vout": 1,
        "scriptSig": {
          "asm":
            "3045022100dd5d33c38abe6f9872c14a3b9f81e505e8c63185ecb0608bb7ddd407144ad
            bba0220278b6cfb12e810d386ca5af9c6293694241f9a20109c3fde723c63c08550fee1[A
            LL] 038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3",
          "hex":
            "483045022100dd5d33c38abe6f9872c14a3b9f81e505e8c63185ecb0608bb7ddd407144ad
            bba0220278b6cfb12e810d386ca5af9c6293694241f9a20109c3fde723c63c08550fee1
            0121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3"
        },
        "sequence": 4294967295
      }
    ],
    "vout": [
      {
        "value": 0.00015400,
        "n": 0,
        "scriptPubKey": {
          "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
            OP_EQUALVERIFY OP_CHECKSIG",
          "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
          "reqSigs": 1,
          "type": "pubkeyhash",
          "addresses": [
            "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
          ]
        }
      }
    ]
  }
}
```

Transaction ID

784d182c2f9421654d8e680593cc3be5582ad925a47a0bcaf15227397e045de2

Signature and Public Key are

483045022100dd5d33c38abe6f9872c14a3b9f81e505e8c63185ecb0608bb7ddd407144ad
adbba0220278b6cfb12e810d386ca5af9c6293694241f9a20109c3fde723c63c08550fee1
0121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3

Signature is

3045022100dd5d33c38abe6f9872c14a3b9f81e505e8c63185ecb0608bb7ddd407144ad
bba0220278b6cfb12e810d386ca5af9c6293694241f9a20109c3fde723c63c08550fee101

Public Key is

038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3

complete key=true

Transaction ID is
784d182c2f9421654d8e680593cc3be5582ad925a47a0bcaf15227397e045de2

Given below is the program output after running it a second time. You can compare the two outputs.

Output

Number of unspent outputs 4

Script public key 76a914d7ce0c2c237ec0ec04931335377a87d16b9865ad88ac

```
{
  "txid": "a6cf2565c1bc667ce65a339234dbd7b43452ec93b9f39caba16eabe4f9ff1adc",
  "hash": "a6cf2565c1bc667ce65a339234dbd7b43452ec93b9f39caba16eabe4f9ff1adc",
  "size": 192,
  "vsize": 192,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "3a0b7cd1e5dd14f5e9c4868914ad7ee9bb6a95f088d381340fd69316dd6fed40",
      "vout": 1,
      "scriptSig": {
        "asm":
          "3045022100bd5dce31b02abdf9088a666352515fe3a62f808fec07b2939c02e571b66443
          4a02200349e792990642c4a4bb6ea3219987b2afffd1662f15c4c9b7ce2892f23af9dc[AL
          L] 038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3",
        "hex":
          "483045022100bd5dce31b02abdf9088a666352515fe3a62f808fec07b2939c02e571b664
          434a02200349e792990642c4a4bb6ea3219987b2afffd1662f15c4c9b7ce2892f23af9dc0
          121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00015400,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
          OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1KFz4ACsUqeFrJsFD2P3YgU7zS4WEs0otv"
        ]
      }
    }
  ]
}
```

```
]
}
Transaction ID a6cf2565c1bc667ce65a339234dbd7b43452ec93b9f39caba16eabe4f9ff1adc
Signature and Public Key are
483045022100bd5dce31b02abdf9088a666352515fe3a62f808fec07b2939c02e571b664
434a02200349e792990642c4a4bb6ea3219987b2afffd1662f15c4c9b7ce2892f23af9dc0
121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3
Signature is
3045022100bd5dce31b02abdf9088a666352515fe3a62f808fec07b2939c02e571b66443
4a02200349e792990642c4a4bb6ea3219987b2afffd1662f15c4c9b7ce2892f23af9dc01
Public Key is
038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3
complete key=true
Transaction ID is
a6cf2565c1bc667ce65a339234dbd7b43452ec93b9f39caba16eabe4f9ff1adc
```

This program sends a transaction to a miner using code written by the Bitcoin Core developers.

Every time this program is executed, the number of unspent outputs reduces by 1. The initial count was 5. To verify, the history command is used and it returns one less output each time.

At the time of first display, the length of the unspendl list is 5, thus indicating there are five unspent outputs. The next time when the program is executed, the value falls to 4 and so on. Thus, the unspent outputs reduce by 1 each time.

The new addition is that the bytes created by the method signrawtransaction are passed to the option sendrawtransaction, which in turn sends the transaction to a miner and returns the transaction id. Also, a minor change introduced here is that the Bitcoins are reduced to 15400.

The option sendrawtransaction returns a transaction id. In the first case, the transaction hash value is 784d. The method signrawtransaction shows the same hash. When these transaction hashes are entered in blockchain.info, they are flagged as valid transactions.

The output and the blockchain explorer shows the same Bitcoin address starting with 1KF. The other Bitcoin address shown is 1Lg. However, when these transaction hashes are checked immediately, blockchain.info makes it very clear that it is an unconfirmed transaction. After some time, the number of confirmations increases. At times, the explorer takes some time before confirming an unconfirmed transaction.

The signature changes each time as the transaction data changes. The public key does not change because the private key does not change.

```
saddr = "1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n"
bitcoins = 0.000154
iaddr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
```

We simply made the two addresses in saddr and iaddr the same. It proves that, the Bitcoins are sent from us to us.

This results in an error, as shown below

```
complete key=true
error code: -26
error message:
66: insufficient priority
```

The error message simply shows the displeasure of the miner. The miner is complaining that he is not being paid enough. The only way out is to increase the miner's fee.

The same error vanishes when a leading zero is added, as the Bitcoin output had more Bitcoins.

bitcoins = 0.0000154

Now, less bitcoins sent to us and more to a miner

Output

Number of unspent outputs 3

Script public key 76a914d7ce0c2c237ec0ec04931335377a87d16b9865ad88ac

```
{
  "txid": "823395396f780e90b2a60eac7484700c6f0e77d0b7be9b715b10f80e6e22acf5",
  "hash": "823395396f780e90b2a60eac7484700c6f0e77d0b7be9b715b10f80e6e22acf5",
  "size": 191,
  "vsize": 191,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "3abd481de741c006728eb2cfd40d20c6753c0420df290fd12090275d2d49a9e2",
      "vout": 0,
      "scriptSig": {
        "asm":
          "30440220182616a21eff49a6d6ffb21f5851e3029abeb8267b22b851976aa450a6584e12
          02205fb564f917d8594b7118425cd3ddfdc0fd0d75dbc3be3ce0742a68ab6693462b[ALL
          ]038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3",
        "hex":
          "4730440220182616a21eff49a6d6ffb21f5851e3029abeb8267b22b851976aa450a6584e
          1202205fb564f917d8594b7118425cd3ddfdc0fd0d75dbc3be3ce0742a68ab6693462b01
          21038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00001500,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 d7ce0c2c237ec0ec04931335377a87d16b9865ad
          OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914d7ce0c2c237ec0ec04931335377a87d16b9865ad88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n"
        ]
      }
    }
  ]
}
```

```
    }  
  }  
]
```

Transaction ID

823395396f780e90b2a60eac7484700c6f0e77d0b7be9b715b10f80e6e22acf5

Signature and Public Key are

4730440220182616a21eff49a6d6ffb21f5851e3029abeb8267b22b851976aa450a6584e

1202205fb564f917d8594b7118425cd3ddfdc0fd0d75dbc3be3ce0742a68ab6693462b01

21038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3

Signature is

30440220182616a21eff49a6d6ffb21f5851e3029abeb8267b22b851976aa450a6584e

1202205fb564f917d8594b7118425cd3ddfdc0fd0d75dbc3be3ce0742a68ab6693462b01

Public Key is

038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3

complete key=true

Transaction ID is 823395396f780e90b2a60eac7484700c6f0e77d0b7be9b715b10f80e6e22acf5

No one complains anymore. And it proves that money is the solution to all insufficient priorities in life. The number of unspent outputs is now down to 3. Also, observe the transaction hash value starting with 8233, it has the same Bitcoin address in the inputs and the outputs.

When the Bitcoin addresses in the source and destination are the same, we are sending Bitcoins to our own Bitcoin address. This is the only change in the program, the value of the saddr variable is now the same as the iaddr variable.

This approach is one of the most common transactions you will meet while strolling along the blockchain. This happens especially when you have a Bitcoin output of say 2 Bitcoins and you need to pay 1.3 Bitcoins to someone else. You create a transaction with just one output of value 1.3 Bitcoins. In effect, you are paying the miner 0.7 Bitcoins for their effort. Any difference between inputs and outputs goes to a miner. The Bitcoin wallet simply creates one more output by paying your bitcoin address 0.699 bitcoins to your own Bitcoin address. This output to itself is called change.

But, when dealing with outputs that carry huge Bitcoin values, this cannot be small change.

Errors When Sending a Transaction Using Code from a Library

```
ch1109.py  
import pybitcointools  
import os  
mybitcoinaddress = '18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7'  
pkey = "KyH6Mm8sYmnT9cm2jMmxpae7CAvbCmmxQ1bNYsjAeMeeT2XD6KWp"  
  
unspend = pybitcointools.history(mybitcoinaddress)  
unspendl = []  
i = 0  
for obj in unspend:  
    if 'spend' not in obj.keys():  
        unspendl.append(obj)  
spendd = unspendl[0]  
print spendd
```

```

outs = [{'value': 30000, 'address': '1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv'}]
tx = pybitcointools.mktx(spendd,outs)
#os.system("bitcoin-cli decoderawtransaction " + tx)
tx1 = pybitcointools.sign(tx, 0, pkey)
#os.system("bitcoin-cli decoderawtransaction " + tx1)
os.system("bitcoin-cli sendrawtransaction " + tx1)

```

Output

```

error: {"code":-26,"message":""64: non-mandatory-script-verify-flag (No error)"}
c7e8034643d029be38218119d48359e859ec0ab16cf18f4e2b4016b303bf165a

```

This second last program shows the perils of writing code using a library that may not be updated for a very long time. In the above program, our own transaction is created using pybitcointools. The above error is displayed.

The program uses the history function to obtain a list of spent and unspent transactions. Then the mktx function is called with the inputs and outputs to create a raw transaction, this function works like the option createrawtransaction. Now, instead of calling bitcoin-cli to sign the transaction, the function sign is used instead. This function signs the transaction.

There is no error up to this point. But when the method sendrawtransaction is executed, the above error code -26 is displayed.

This problem or the error did not occur the first time we ran the code. The transaction hash value starting with c7e8 is displayed for you to check in a blockchain explorer.

A Google search didn't give zillion replies but along the way, we learnt that the pybitcointools library must be downloaded from:

<https://github.com/wizardofozzie/pybitcointools>.

As advised, the file was downloaded and unzipped, and then in very same folder, we give the following command:

```
$sudo python setup.py install
```

Or follow the pip approach.

Using the Bitcoin Library to Send Transactions

```

ch1110.py
import bitcoin
import os
mybitcoinaddress = '18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7'
pkey = "KyH6Mm8sYmnT9cm2jMmxpae7CAvbCmmxQ1bNYsjAeMeeT2XD6KWp"
unspend = bitcoin.history(mybitcoinaddress)
unspendl = []
i = 0
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)
spendd = unspendl[0]
outs = [{'value': 30000, 'address': '1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv'}]

```

```
tx = bitcoin.mktx(spenddd,outs)
tx1 = bitcoin.sign(tx , 0 , pkey)
os.system("bitcoin-cli sendrawtransaction " + tx1)
```

Output

```
23df8c07a0f5b247ba69a02638791e7a1513ca5a24e440e318eed32e86f8d996
65588f0746789429b8635939e34f9471149582ec2734484d4d64e4058d6280db
```

To verify if everything works as advertised, we run the code twice and still no error. Everything remains the same, except one; the import from pybitcointools modules is replaced with the bitcoin module. The Bitcoin addresses and private key are the same as used in the past. The code using this library is much cleaner than the earlier libraries. Finally, the transaction hashes display our Bitcoin address in a blockchain explorer, which reconfirms that these transactions were made by our code.

To narrate our experiences:

We tried using other blockchain technologies like those created by J P Morgan and Intel. They insisted that we run these blockchains in a program called vagrant. We download the software and installed it on our Mac. The program refused to run and gave us a SSL error. We searched the net and tried installing an older version of SSL. Low and behold, the history function failed. We must be unlucky, you would say. When you install some software, some other unrelated software on the computer stops working. One of the many reasons why we show you real output; you can cross check with any blockchain explorer. If code does not work, remember you are in very good company.

At times, we have also had very unusual errors and patience is the word here.

Let's take a very long detour and explain why we lost most of our hair in the last two years while learning Bitcoins and then writing this book.

In the last century, we only used Windows. Our favorite book of all times was Undocumented DOS. Now, we only use a Mac with Linux running in a virtual machine. We started with an iMac. Today on a day to day basis we use 2 machines, a Retina iMac at home and a Mac Pro at work. You read this right, the cylinder as a computer with two monitors. Yes, we are showing off.

My advice to Apple, only sell machines with dual monitors. We also have a Mac Book Pro as a backup laptop in the office.

For a long time, everything chugged on merrily, thing did work as advertised. We wrote our chapters, placed everything in a Dropbox folder to share it between resources.

We made sure that we double checked everything. The only reason being that we did not trust our knowledge of Bitcoins. Our motto was, if blockchain.info agrees with us, only then we are right. The blockchain explorer can never be wrong, we can and will.

We then started editing the book and checking the programs before sending it to our publisher. We tried every program and corrected the grammatical and technical errors.

Then disaster struck. The history function of the earlier chapter gave us a SSL error on our office computer. This history function is not something we wrote, it a function written by the main designer of Ethereum.

To add to this misery, the same function worked at home. There must be something we installed on the office computer. Normally both are computers are in sync when it comes to software installed. We took the easy way out and simply used a backup that was a 3-month-old. Fortunately, everything now worked.

Then lighting struck again. This chapter, last program. We now get an error Missing inputs. All that Bitcoin core is telling

us that you have some memory issues with transactions. We used a method, explained in chapter 22, on how we got this resolved.

In all this happening around us, we get one more error. Cannot connect to server. Now we hit the nail on the head, the Bitcoin server is down. Before we went to town announcing that Bitcoin is broken, we realized that we forgot to run the bitcoin daemon.

Our advice, if things do not work, take a deep breath and start from scratch with a clean machine. We keep installing all sorts of software and things just break.

We only could write so much because we kept compiling and recompiling the C++ source code that Bitcoin is written in. This was our primary documentation. Google came next.

The situation is that the same code that we compiled some zillion times now does not execute on both the machines at work and in the office. They however, work as advertised on our laptop and old iMac. Too cynical to ask why. We will show you the error messages when we come to the Bitcoin source chapter.

We solved this problem by simply reinstalling the Mac Operating System OSX.

Beware, thou have been warned.

CHAPTER 12

Client and Server

This chapter focusses on the interactions between the client and the server. A bitcoin client speaks exclusively to a bitcoin server. The bitcoin server does all the heavy lifting work. Here, we have bitcoin-cli for the client and bitcoind for the server. The two programs, bitcoind and bitcoin-cli, are part of the Bitcoin codebase. As Bitcoin is open source, this code can be downloaded off the Bitcoin website.

```
$bitcoin-cli getblockcount
```

Output

```
error: couldn't connect to server
```

The error is thrown because the bitcoin server is down, it must be started. In the last chapter, we executed the program bitcoind as

```
$bitcoind -daemon
```

Now, we run it as

```
$bitcoind -printtoconsole
```

On the screen, you see lots and lots of lines flow past. The important lines however, are

Output

```
Default data directory /Users/vijaymukhi/Library/Application Support/Bitcoin  
Using data directory /Users/vijaymukhi/Library/Application Support/Bitcoin  
Using config file /Users/vijaymukhi/Library/Application Support/Bitcoin/bitcoin.conf  
Verifying last 288 blocks at level 3
```

The output fills up the entire terminal window of which some messages are useful, some not so. The command line parameter printtoconsole makes the bitcoin server more verbose and talkative. It gives information on the commands, directories and files used by the server when it starts.

Now when we run the first command in another window, we see the following output on the screen.

```
$bitcoin-cli getblockcount
```

Output

```
488273
```

The option getblockcount gives a count of the total number of blocks downloaded from the miners on our disk. Your mileage will however vary a lot.

```
$bitcoin-cli getblockhash 0
```

Output

000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

The `getblockhash` command gives the block hash value of the given block number. Here, we have the hash value of the genesis block.

```
$bitcoin-cli getblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

Output

[illegible]

Once we have the hash value, we use the `getblockhash` command to acquire all the block data without going to website `blockchain.info`.

```
$bitcoin-cli getblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f false
```

Output

0100
3ba3edfd7a7b12b27ac72c3e67768f617fc81bc3888a51323a9fb8aa4b1e5e4a29ab5f4
9ffff001d1dac2b7c01010000000100
000000000000000000fffbfff4d04ffff001d0104455468652054696d6573203032f4a6
16e2f32303039204368616e63656c6c6f72206f6e206272696e6b206f66207365636f6e64
206261696c6f757420666f722062616e6b73ffffff0100f2052a01000000434104678afdb
0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61deb649f6bc3f4cef38c4
f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5fac0000000

The parameter `false` displays the same output as a series of bytes instead of a verbose one. The command `getblock` takes a block hash value and not a block number.

```
$bitcoin-cli getdifficulty
```

Output

209453158595.381

There are numerous command line arguments like `getdifficulty`, `help` etc. which can be given to the client. The client program `bitcoin-cli` sends these parameters to the bitcoin server using standard Internet networking protocols and then waits and waits for the server to return an answer. Here, the `getdifficulty` returns the current difficulty, this value also changes with time.

```
$bitcoin -help
```

All the options that can be given to the bitcoin server are displayed with the `help` parameter. These are options and not commands, which in our view are more useful.

```
$bitcoin -printtoconsole
```

The command line options passed to the bitcoin server must be accurate. It ignores the option if it is not valid. An extra `s` is added to `printtoconsole`. The bitcoin server returns no error messages and no output.

```
$bitcoin -printtoconsole -checkblocks=2
```

Output

```
Verifying last 2 blocks at level 3
```

The bitcoin server on start, checks or verifies the last 288 blocks before allowing the client to use its web functionality. At times, this startup takes too much time. The command line option `-checkblocks` is given several blocks which the server should verify. The default is 288. If the value is 0, then all the blocks are verified, which takes a million years.

```
$bitcoin-cli stop
```

Output

```
Bitcoin server stopping
```

The `stop` option stops the server. It is advisable to stop the client when it is not needed as you stand a very good chance of corrupting your blockchain. And if the blockchain is corrupted, it takes days and days to reindex.

The million-dollar question is how stringently does the bitcoin server verify blocks.

```
$bitcoin -printtoconsole -checkblocks=2 -checklevel=0
```

Output

```
Verifying last 2 blocks at level 0
```

The parameter `checklevel` decides on the stringent checks performed on every block. The default value is 3. There are 5 possible levels, starting from 0 to 4.

Level 0 – it checks if the block can be read from disk. This takes the least amount of time to accomplish.

Level 1 – the check is further to validate the block. Validity is basically having no duplicate transactions, mismatches on merkle hash values, the size of the block being less than 1 Mb, the first transaction to be a Coinbase transaction, etc. Rigorous checks for just one jump in level.

Level 2, - it verifies the undo block. The verification is on the hash value of the undo block. Some of this may sound tricky right now, but it will get clear by the end of the book.

Level 3 – there is an extra check on the consistency of the utxo dataset. This database resides in the `chainstate` folder of the Bitcoin folder. The utxo or unspent outputs is a database of all outputs that have not yet been spent. The heart and soul and everything of Bitcoins.

Level 4, is the most in-depth one. Here, the check is on the cryptographic signatures of the inputs, they must match. This is the slowest of all verifications and should not be used every day. One of the many reasons is that Elliptic Curve Cryptography is very slow.

```
$bitcoind -datadir=/Users/vijaymukhi/zzz
```

Output

```
Error: Specified data directory “/Users/vijaymukhi/zzz” does not exist.
```

The datadir option is given to decide where the bitcoin data or blockchain will be stored. The folder must be created in advance.

```
$bitcoind -datadir=/Volumes/Verbattin/Bitcoin -conf=/Volumes/Verbattin/Bitcoin/bitcoin.conf -printtoconsole
```

At times, because of the sheer size of the blockchain, you may prefer a slower external drive. We have an external USB drive called Verbattin on which we create a folder Bitcoin. The Bitcoin server will now use this folder as the root folder for storing all its dossiers.

The downside is that the default location must be specified each time, but there is a way to override it.

The -conf parameter informs the bitcoin server to look for the configuration file bitcoin.conf but in another location and not the default one. We choose the new Bitcoin folder. The name of the file can be changed also. So now when the bitcoin server starts, it reads a different conf file present on the USB Verbattin drive.

```
Verbattinn\Bitcoin\bitcoin.conf
rpcuser=bitcoinrpc
rpcpassword=mukhi
testnet=1
```

One extra line is added; the key testnet is set to 1. By default, Bitcoin uses the mainnet network which is the paid network. We have been using this network all this time. In the testnet network, we can now freely spend Bitcoins as they have no value outside the testnet network.

The new folder is called testnet3 and it has an identical blocks folder structure with the same blk and rev file names. Everything remains the same, except that when the bitcoind server starts, it loads the testnet blockchain on the external USB drive. For experimental purposes, most people use the testnet network as the Bitcoins are not real. Bitcoins on the testnet network have no value at all.

Let's now find the blockhash of the genesis block.

```
$bitcoind -printtoconsole -checkblocks=2
```

This command is executed on the mainnet server. Now, there are two bitcoin servers running at the same time and we can interact with both at the same time. This is possible because the servers listen to connections on different ports.

The mainnet server refers to the old bitcoin.conf file and the old default data location.

```
$bitcoin-cli getblockhash 0
```

Output

```
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

This is the good old genesis block hash.

```
$bitcoin-cli -testnet getblockhash 0
```

Output

```
000000000933ea01ad0ee984209779baaec3ced90fa3f408719526f8d77f4943
```

The option `-testnet` directs the bitcoin client to connect to the testnet bitcoin server. The testnet server downloads the entire testnet data or blockchain.

Let's now create a Bitcoin addresses. Just a quick reminder, that all mainnet or real Bitcoin addresses start with a 1.

```
$bitcoin-cli getnewaddress
```

Output

```
1EXDDdArc25DopBDCrhUMiJDHfzNqkNqev
```

```
$bitcoin-cli -testnet getnewaddress
```

Output

```
mytT6gidgr4AmkdN71gKCIEEcjrmvcasrP
```

The testnet addresses start with a m or a n.

```
$bitcoin-cli -testnet getaddressesbyaccount ""
```

Output

```
[  
  "maggPrtjRruTFFpgmp1iHjSFhnyZ1Zn1FjP",  
  "mvDmuGkZ296RPDEgZwMHFUdLNuo5CGQAVV",  
  "mytT6gidgr4AmkdN71gKCIEEcjrmvcasrP"  
]
```

All that works with the mainnet works seamlessly with the testnet network. However, a few things like Bitcoin address prefixes, private key prefixes change. A Bitcoin address starting with a m or a n are worthless. Never accept it.

```
$bitcoin-cli getbestblockhash
```

Output

```
00000000000000000000000028b308825a173354eb3c9bc57d907d805c6411931144e8e
```

Block 425577 has this block hash.

Generally, it takes 10 minutes to create a new block. At times, it takes more time. For example, in 2011 the time between block 152218 and 152217 was 1 hour and 29 minutes. Block 152217 was created at time 06:09:50 on 2011-11-07. Block 152218 was created on the same day but the time was 07:48:57.

After a couple of minutes, we run the same command again.

```
$bitcoin-cli getbestblockhash
```

The result is as follows.

Output

```
0000000000000000000000003bc393206328d58b161e5f98cd7270434aaf2a3d6b92c14
```

It looks like we waited too long therefore a block got skipped. The hash is of block 425579 and not 425578. The time difference between blocks 425579 and 425578 is only 2 minutes. When we re-tried the above command, at the time of editing our book, it took way too long for a new block to be downloaded, so we used the older values.

The option `-dbcache` sets the internal cache size. This cache is used by Bitcoin for caching the leveldb databases. Changing these values showed no significant change in performance whatsoever.

You can visit websites like <https://tpfaucet.appspot.com> where you can get free testnet bitcoins. You can use google to guide you to other testnet faucets as well. You also need to give back the Bitcoins once you have used them. It is however only a gentleman's promise. We will revisit the testnet network exclusively in a later chapter.

You can either use command line parameters with the `bitcoind` program or place these parameters in the configuration file, `bitcoin.conf`.

Always use the bitcoin client to stop the bitcoin server. The bitcoin server caches all the Bitcoin blocks and databases so if there is no graceful exit, the next time around when the bitcoin server starts, it will take a very long time. The only way to shut down the bitcoin server gracefully is by running the following command.

```
$bitcoin-cli -testnet stop
or
$bitcoin-cli stop
```

Please do not use the OS's kill command.

The mainnet and testnet are very similar. The third type of network is called regtest. This is your own private bitcoin network, so no connecting to peers.

We first shut down all the other running bitcoin servers and then run

```
$bitcoind -regtest -printtoconsole
```

No one in the world can access this private Bitcoin network. On our Mac, there is a new folder called regtest created in the default Bitcoin folder,

```
/Users/vijaymukhi/Library/Application Support/Bitcoin
```

It looks very similar to the real Bitcoin network as it has a folder called blocks. There is only one file `blk00000.dat` which is 16MB large and an empty `rev` file. If you open the `.dat` file in a hex editor, you will see just one block and it starts with the magic number `FA BF B5 DA`. The size of the block is 285 bytes.

```
$bitcoin-cli -regtest generate 1
```

Output

```
[
    "17e37daa125d718ce20a7f25cf6b6cafad48f735676c048a4e13beaa027a87e9"
]
```

The generate command creates a Bitcoin block. The number of blocks to be generated depends on this value, here we have asked for just one block. To verify the same, in the hex editor, jump to position `285 + 8` in the file `blk00000.dat`. You will see one more block added with the same magic number.

```
$bitcoin-cli -regtest getbalance
```

Output

```
0.00000000
```

There is a perception that Bitcoins are freely given while creating a block; it is simply idle talk. The get balance command returns a balance of 0.

```
$bitcoin-cli -regtest generate 98
```

The above command creates 98 more blocks but the balance is still 0.

```
$bitcoin-cli -regtest generate 1
```

We try again, but no luck.

```
$bitcoin-cli -regtest generate 1  
$bitcoin-cli -regtest getbalance
```

```
Output  
50.00000000
```

Finally, there is a balance of 50. After 101 block, there is a balance of 50 BTC. This suggests that a block needs 101 confirmations to claim the Bitcoins.

```
$bitcoin-cli -regtest generate 1
```

```
Output  
[  
  "72153bb5ab8f158e849b185bf003b6f6ce9e9cb9c2443c9c297e37fc98cc8c66"  
]  
$bitcoin-cli -regtest getbalance
```

```
Output  
100.00000000
```

With every new generation of a block, 50 Bitcoins are added. However, the bad news is that only the first 150 blocks generate Bitcoins.

```
$bitcoin-cli -regtest getnewaddress
```

```
Output  
mm9Ct1H411uJHab7A5eF XuNij8Qg6aJBHg
```

The addresses created in regtest are like testnet, they do not start with a 1.

```
$bitcoin-cli -regtest getaddressesbyaccount ""
```

```
Output  
[  
  "mg3Wj5gtxCdogwtnpNsotvzpBsvE7Do8eX",  
  "mm9Ct1H411uJHab7A5eF XuNij8Qg6aJBHg"  
]
```

The three networks work in the same fashion, mainnet is the real thing, testnet is like mainnet but the Bitcoins have no value and regtest create a private network.

The problem with the testnet network is that it is very unpredictable with respect to time. At times, it takes forever for a block to be confirmed. Also, there is no money or incentive for anyone to run a testnet network. But with the rising prices of Bitcoins, testnet is the way to go.

Let's look at nearly every file or folder present in the main Bitcoin folder. This is only a short and gentle introduction to what lies ahead.

Our two sources for the explanations here are the Data Directory topic/section from the bitcoin wiki and the files.md file from the Bitcoin sources.

The book is very Mac centric as all that we own is a Mac and Linux running on a virtual machine. On Windows Vista, or a higher windows OS, the default data directory is C:\Users\YourUserName\AppData\Roaming\Bitcoin. On Linux, it is ~/.bitcoin/. The option -datadir changes the default location to a new one, but do not change it. In the default Bitcoin folder, please run the following command as in:

```
$ls -al
```

The equivalent in Windows OS is the command `dir /a`.

This command displays all the hidden files. A file called .lock is now visible; the dot indicates it is a hidden file.

For some reason, Bitcoin uses two different databases leveldb and BerkleyDB to store key value pairs. These databases are single user; leveldb uses a folder to store the key value pairs and BerkleyDB uses a single dat file. There is no reason why Bitcoin then and now uses two different databases.

The leveldb database uses a file called LOCK in the leveldb folder to denote that another instance is using the database. To be different, BerkleyDB uses a file called .lock. Both files are 0 bytes large.

The bitcoin.conf files has configuration details to change the behavior of the Bitcoin server. Most of these key value pairs can also be passed as command line arguments to the bitcoind server and bitcoin-qt. The name of this file can also be changed as well.

The blocks folder.

The blkxxxxx.dat files contain the actual Bitcoin blockchain. When a block is downloaded, this block file is processed and the Bitcoin blockchain is extended. These blocks are mostly ignored unless a re-indexing is done. The massive size of the Bitcoin folder is mainly due to these blk files. The blocks folder is over 200GB in size and increasing.

These files are deleted when the bitcoin data directory is pruned. So, caution advised. There is no undo command to undo the delete and the blk files must be downloaded again. The file size of each of these files is about 128MB.

There is no rational order while storing blocks in the blk files. So, block 100 may not be stored before block 99. Then there are blocks which are not linked to other blocks as they are orphans and not important.

The data in the blk file changes computer to computer, but all have a valid copy of the Bitcoin data. To maintain a similar copy of the blockchain from one computer to the other, the entire Bitcoin folder is copied and not just one folder. The reason being, all Bitcoin directories are created using the blk files as a base. The indexes or rev files will not match if only a few files or folders are copied.

The rev files contain the undo data. These files are around 18MB in size and hence the undo data is compressed. An entire chapter in the book is allotted to understanding this file format. For example, if block 123 is in file number 125, then the undo data for this block will be found in rev file rev00125.dat. This file format is only documented in the source code and not on the Internet. We tried googling but had no success.

The size of the index folders depends upon the Bitcoin settings on the computer. This folder enables positioning the file pointer directly at the starting point of the block in the blk and rev files.

This is a leveldb database. If transaction indexing is enabled, then there are over 150 million key value pairs giving similar information as to where a transaction begins in the blk files. Transaction hashes are not stored in the rev files. There is a separate chapter to explain what is stored in this folder.

Bitcoin is termed to be complicated because it is very simple. There is no running total of money stored in the Bitcoin addresses, there is no state information stored in the transaction. They are all independent of each other. However, there is a way to figure out which outputs have some unspent money.

Let's assume Bitcoin address 1ASD is the owner of 10 Bitcoins. This data is stored in the utxo database which is a leveldb database in the chainstate folder. When these 10 Bitcoins are spent, the transaction having the address 1ASD is removed from the utxo database. The chain state folder is about 1.8 GB large. It deserves more than one chapter to itself. The blockchain is the undo data set.

For users, the most important file is wallet.dat. It stores all the private keys and many more things. This is the only file that must be backed up as per the Bitcoin guidelines. This is however a BerkleyDB database. A very small file around 119KB. The database folder, if present, is used by the BerkleyDB database environment for storing journaling files, but only for the wallet. The size is approximately 1MB. One of the forthcoming chapters is on BerkleyDB database.

In Bitcoin version 0.15, a file called mempool.dat is introduced. We will cover this file format on a rainy day.

There are two log files, db.log and debug.log. The db.log file is small around 17k. It reveals if the wallet was opened and closed. Nothing useful. On the other hand, the file debug.log is 107MB large. It stores the output of the command option printtoconsole. The wiki calls it the Bitcoin verbose log file. The file is constantly trimmed as both bitcoind and bitcoin-qt write to this circular log file and the file size can become very large.

```
$ps -e
```

Output

```
590 ttys002 723:35.41 bitcoind -printtoconsole -checkblocks=3 -checklevel=0
```

The command ps -e shows all the processes running on our Mac. Here, the program bitcoind is running with a pid or process id of 590.

```
$cat bitcoind.pid
```

Output

```
590
```

The file bitcoind.pid displays the process id of the running Bitcoin server. Every time the bitcoin server, bitcoind or bitcoin-qt starts, it looks at this folder for a file called bitcoin.pid. If this file is found, then it recognizes that an instance of Bitcoin is already running. If not then the server instance is started and this file is created.

Two instances of Bitcoin cannot be running at one time. The Bitcoin server creates this pid file at startup and then deletes the file at shutdown. If you switch your machine off while Bitcoin is running, this file does not get deleted on its own, it yet exists.

Most of the subsequent chapters are written after reading the Bitcoin source code.

There is a chapter in the book for the following three files.

The file peers.dat contains all the peers or bitcoin servers or miners who sends the blocks. The fee_estimates.dat gives statistics to determine the minimum amount of money to pay miners for mining the transactions. Nodes that are banned are stored in banlists.dat.

Then there are files like onion+private+key which are used with Tor for anonymous connections. On a Mac, Tor is a pain in the neck to use.

There is a chapter in the book that attempts to unsuccessfully define a bitcoin and a blockchain. Most people who do not understand bitcoins think that the blk files are the blockchain.

CHAPTER 13

Notaries and OP_RETURN

This chapter elucidates a different aspect of the blockchain ecosystem, as a notary. The Economist called the blockchain, a trust machine. Once something is stored in the blockchain and there are seven confirmations, then no one can dispute its existence or validity. The above line of having seven confirmations is literally taken from multiple sites on the Internet. Our first understanding was that seven miners must confirm the same transaction. However, it implies that one miner has finally managed to calculate the sha256 hash value correctly. This process is called mining and a transaction is included in a block only after it is successfully mined.

Let's add a block, say A to the blockchain, it has been mined successfully. After some time, a new block B is mined. The previous block hash field of block B contains the hash of the block A and the new block is added to the blockchain. This process takes about 10 minutes on an average. Block B and not Block A, is now the last block or at the tip of the blockchain. Now another block C, gets confirmed. This block now sits in front of block B, like a stack of plates. At this point, there is one more confirmation. Then after 10 minutes another block, D gets confirmed and there will be three confirmations. The more the confirmations the greater the probability that this chain of blocks will not get replaced by another parallel chain.

Now in the Bitcoin peer to peer network, miners are continents apart. It takes some time for this information, on which miner won, to reach everyone on the network. It will happen that some miners will add the block A to their copy of the blockchain, and some miners would add the block C. It could so happen that another miner also figured out the Sha256 hash value say seconds after the first. This rarely happens, but it does happen.

In such cases, the blockchain splits and there are more than one chains.

Nevertheless, at some point in time, the problem is resolved. The chain that wins is that chain which has the largest number of blocks. After some time, a steady state is reached where we come back to just one blockchain. Also, it is myth that it takes just over an hour for seven confirmations to take place. At times, it takes three hours if the workload is heavy. These days the workload is very heavy.

You can use a trick here. There is a chance that if you leave a bigger tip or fee for the miner, the number of confirmations will increase quickly and your transactions will be included faster than others.

Let's look at some code now.

Storing Non-Bitcoin Transaction Data, the Right Way in the Blockchain

```
ch1301.py
import bitcoin
import subprocess
import os
import ast
from cfuncs1d import *
```

```
saddr = "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
bitcoins = 0.0000154
iaddr = '18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7'
unspend = bitcoin.history(iaddr)
unspendl = []
i = 0
if len(unspend) == 0:
    print "No Outputs available to spend"
    exit(0)
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)
print "Number of unspent outputs %d" % len(unspendl)
spendd = unspendl[0]
newobject = spendd['output']
ind = newobject.index('.')
thash = newobject[0:ind]
outi = int(newobject[ind + 1:])
scr = scrpubkey(thash , outi)
str0 = '[{"txid": "%s", "vout": %d}]' % (thash , outi)
str1 = '{"%s" : %f, "data": "56696a61794d756b6869"}' % (saddr , bitcoins )
raw = subprocess.check_output(["bitcoin-cli", "createrawtransaction", str0 , str1 ])
str2 = raw[:-1]
str3 = '[{"txid": "%s", "vout": %d, "scriptPubKey": "%s", "redeemScript": "%s"}]' % (thash , outi , scr)
pkey = 'KyH6Mm8sYmnT9cm2jMmxpae7CAvbCmmxQ1bNYsjAeMeeT2XD6KWp'
str4 = '["%s"]' % pkey
raw1 = subprocess.check_output(["bitcoin-cli", "signrawtransaction", str2 , str3 , str4 ])
raw1 = raw1.replace('true', '"true"')
outdict = ast.literal_eval(raw1)
rawoutput = outdict['hex']
os.system("bitcoin-cli decoderawtransaction " + rawoutput)
raw2 = subprocess.check_output(["bitcoin-cli", "sendrawtransaction", rawoutput])
print raw2
str = "VijayMukhi"
for i in str:
    print "%x" % ord(i),
Number of unspent outputs 3
{
    "hex":
    "0100000001f5ac226e0ef8105b719bbeb7d0770e6f0c708474ac0ea6b2900e786f399533
    82000000006a47304402202007bdc8eb79376d21cc72ac9991a3da1dc13dcfb635525545
    ceb783c520383602200d6a0230a7f7d962f831d105ac7407c080145dac33ceb561207714
    05e04e9dda0121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd36
    5774dce3ffffff02e8030000000000001976a914c847b6d84ecf8048474b19c4a2330409
    ff32a1ad88ac0000000000000000c6a0a56696a61794d756b686900000000",
    "complete": true
}
```

```

{
  "txid": "90154afd25089357f1cc7f9bd300781f903a1afec5a1e19ae189b902fca84b6b",
  "hash": "90154afd25089357f1cc7f9bd300781f903a1afec5a1e19ae189b902fca84b6b",
  "size": 212,
  "vsize": 212,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "823395396f780e90b2a60eac7484700c6f0e77d0b7be9b715b10f80e6e22acf5",
      "vout": 0,
      "scriptSig": {
        "asm":
          "3044022007bdc8eb79376d21cc72ac9991a3da1dc13dcfb635525545ceb783c520383
          602200d6a0230a7f7d962f831d105ac7407c080145dac33ceb56120771405e04e9dda[AL
          L] 038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3",
        "hex":
          "473044022007bdc8eb79376d21cc72ac9991a3da1dc13dcfb635525545ceb783c5203
          83602200d6a0230a7f7d962f831d105ac7407c080145dac33ceb56120771405e04e9dda0
          121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00001000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
          OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
        ]
      }
    },
    {
      "value": 0.00000000,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_RETURN 56696a61794d756b6869",
        "hex": "6a0a56696a61794d756b6869",
        "type": "nulldata"
      }
    }
  ]
}

```

```
    }  
  ]  
}
```

Output

```
90154afd25089357f1cc7f9bd300781f903a1afec5a1e19ae189b902fca84b6b  
56 69 6a 61 79 4d 75 6b 68 69
```

For these programs to work as advertised, the Bitcoin address in variable `iaddr` and the private key stored in variable `pkey` must be changed by you.

So far, we have been using the Blockchain to store only Bitcoin transactions data but as shown in one of the earlier chapters, people have stored emails also. Most of the fields in the Blockchain are fixed in size except for the input script and output script.

The input's `scriptSig` field supplies the data for unlocking the Bitcoins of an output. So, the `scriptPublicKey` of the output uses this `scriptSig` as the key or data to unlock it. No one can tamper with the script of the inputSig, it's a different story with the output `scriptPublicKey`. The outputs give information on the destination addresses of the Bitcoins, the inputs give the source of the money. The rules set by the outputs are to be strictly followed and not inputs. That's because the outputs carry code, the inputs have only data.

The miners are simply interested in the Bitcoins they make per transaction. The Bitcoin world has no qualms if there is a Bitcoin value of 0 in the output. There is a lot of junk stored in the outputs as well and nobody cares. The Bitcoin community legally allows you to store anything you want. To demarcate the important data from the other, there is a script opcode called `0x6a` which is known as `OP_RETURN`. The word `return` in any programming language means stop execution immediately. On the same lines, when the Bitcoin scripting engine sees this opcode, all script processing stops immediately.

For example, if we create an output `scriptPublicKey` starting with the `OP_RETURN` opcode, everything thereafter is ignored by the miner's Bitcoin script engine. The transaction is still a valid transaction.

We bring in an earlier code that we wrote and make only two modifications. First, there is an error check to ensure that there is no further processing if there are no outputs to spend. This is determined by the length of the `unspend` list. Its length must not be 0.

Next, a key called `data` is added to the string variable, `str1`. This variable supplies the outputs to the transaction. The value of this `data` key name is a set of simple hex bytes presently having the hex bytes of my name. The program displays the hex bytes of my name towards the end. There are no other changes.

This program in its native form may display an error if you are using Bitcoin Core version 0.12.0. The same code will work seamlessly from version 0.13 onwards. But, if you are working on a lower version, the code will simply not run.

When the program is executed with the `decoderawtransaction` option, it shows one input but two outputs.

The first output as before, is for transferring 00001000 BTC to ourselves. The BTC value is reduced to get a faster level of service from the miners.

The second output is a bit of a surprise. When the program, `bitcoin-cli` spots a key called `data`, it creates one more output. The `scriptPublicKey` of this output starts with a value of `6a`, which is the opcode for `OP_RETURN`. Then, it counts the number of bytes assigned to the `data` key. In our case, it is 10. Therefore, it places 10 or `0x0a` as the second byte. Finally, it adds the 10 hex bytes given as the value to the key `data` and adds them to the end of the key, `scriptPublicKey`.

The amount in Bitcoins is 0. Only the data is passed as the bytes prior to it are placed by the client code, bitcoin-cli. The website blockchain.info displays an error initially and marks it a non-standard transaction.

The error we get is:

Transaction rejected by our node. Reason: None Standard Script Output OP_RETURN 56696a61794d756b6869

When this non-standard transaction gets confirmed, the website blockchain.info also accepts it and displays it at the very end:

```
OP_RETURN 56696a61794d756b6869
(decoded) j VijayMukhi
```

However, blockchain.info is not totally convinced. It still displays this output, strangely, in red. It must be noted that the Bitcoin value is 0. Again, anything placed after an OP_RETURN opcode is ignored as the script engine does not read code following thereafter.

There is no Bitcoin address in this output and the Bitcoin value is also 0. In effect, this output is taking up unnecessary space in the Blockchain. But since the miner is paid his dues to add the transaction to the blockchain, he does it dutifully and nobody complains either.

There is a practical application though from security point of view. If we have signed a digital legal document, or a scanned pdf file, the double SHA256 hash adds more security. The hash value will be safely stored in the blockchain, starting with an opcode of OP_RETURN and followed by the length of the hash 32 bytes. No one can tamper with this file ever. This fact proves that this document, signed by a few people, existed at some point in time. It is not debatable. Also, anyone having a copy of the document can verify the SHA256 hash value and verify the contents are original.

Not Hardcoding the Private Key in the Code

```
ch1302.py
import bitcoin
import subprocess
import os
import ast
from cfuns1d import *
rawadd = subprocess.check_output(["bitcoin-cli", "getaddressesbyaccount", ""])
print rawadd
outlist = ast.literal_eval(rawadd)
print outlist[0]
rawpriv = subprocess.check_output(["bitcoin-cli", "dumpprivkey", outlist[0]])
print rawpriv
pkey = rawpriv[:-1]
saddr = "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
bitcoins = 0.0000154
iaddr = outlist[0]

unspend = bitcoin.history(iaddr)
unspendl = []
i = 0
if len(unspend) == 0:
    print "No Outputs available to spend"
    exit(0)
```

```
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)
print "Number of unspent outputs %d" % len(unspendl)
spendd = unspendl[0]
newobject = spendd['output']
ind = newobject.index(':')
thash = newobject[0:ind]
outi = int(newobject[ind + 1:])
scr = scrpubkey(thash , outi)
str0 = '[{"txid": "%s" , "vout": %d}]' % (thash , outi)
str1 = '{ "s" : %f , "data": "56696a61794d756b6869697361656d626563696c65616e64616666f6f6c" }' %
(saddr , bitcoins )
raw = subprocess.check_output(["bitcoin-cli", "createrawtransaction", str0 , str1 ])
str2 = raw[:-1]
str3 = '[{"txid": "%s" , "vout": %d , "scriptPubKey": "%s" , "redeemScript": ""}]' % (thash , outi , scr)
str4 = '[ "%s" ]' % pkey
raw1 = subprocess.check_output(["bitcoin-cli" , "signrawtransaction", str2 , str3 , str4 ])
raw1 = raw1.replace('true' , ""true")
outdict = ast.literal_eval(raw1)
rawoutput = outdict['hex']
raw2 = subprocess.check_output(["bitcoin-cli" , "sendrawtransaction", rawoutput])
print raw2
str = "VijayMukhiisaembecileandafool"
s = ""
for i in str:
    s = s + "%x" % ord(i)
print s
```

Output

```
[
    "18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7",
    "1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74",
    "1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7",
    "1DTGvGaFHfeHvbZThZrPbddM4KgF6E4QHn"
]
18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7
KyH6Mm8sYmnT9cm2jMmxpae7CAvbCmmxQ1bNYsjAeMeeT2XD6KWp
Number of unspent outputs 25
d2d9cb2f681c5d0c93cc103476b57e94f15a25a84b35d04a0dcd8b562a570c04
56696a61794d756b6869697361656d626563696c65616e64616666f6f6c
```

This program behaves like the earlier one but there are a few modifications made to it. It is more generic. The variable names remain the same but they are not initialized to a constant value.

For example, to obtain the addresses, the option `getaddressbyaccount` is executed with a null string. The return value is a list comprising of valid Bitcoin addresses created by our wallet. The list has four addresses. Your mileage will vary.

The variable `rawadd` is a simple string. The `literal_eval` function creates a list called `outlist`. The first member of this list is the Bitcoin addresses used in the past. These addresses must have bitcoins to spend. Thus, there is no need to assign a Bitcoin address manually, simply choose the first Bitcoin address from the wallet.

Now, let's acquire the private key that created this Bitcoin address.

The method `dumpprivkey` returns a string that represents a private key. It is this private key that generated this Bitcoin address. All this time, we have been crying from rooftops that given a public Bitcoin address, you cannot get the private key. We yet stand by what we have said. The Bitcoin wallet creates this Bitcoin address and therefore it knows the private key. The wallet's job is to store this private key internally. The wallet also uses this private key to sign the transactions.

The problem with these strings is that they normally contain an enter sign at the end. So, the slice operator `[:-1]` is used to remove this pesky enter.

It is not compulsory to key in a Bitcoin address that has ownership of a few Bitcoins nor it's private key. You are free to decide which Bitcoin address should get the bitcoins. However, if there is only one Bitcoin address in the wallet, then the choices are limited.

We have given some text for the world to recognize us. The data bytes describe me J. This data is now stored in the blockchain for eternity, courtesy blockchain.info.

```
OP_RETURN 56696a61794d756b6869697361656d626563696c65616e6461666f666c
(decoded) jVijayMukhiisaembecileandafool
```

Please check the transaction hash beginning with `d2d9` shown in the output in blockchain.info.

All the apps that store data in the blockchain use the `OP_RETURN` method. Hashes of images, videos, documents etc. are saved in the Blockchain. It proves with certainty the existence of the digital document.

The next example adds some data into an output but the transaction is not spendable. This is a very cheap way of adding things to the blockchain.

Let's first key in this special transaction hash value in blockchain.info.

```
29b781831f30f1f8d91aba3fb9996000b0c62b4c49aab7d10b6d0bc97754b3b1
```

The website blockchain.info shows the following

```
Output Script 6a 20 e36c7d4102bb43c8ec3748f23b62c484c5ea7e00c4ea51e6d49743349311f22a
```

The Output script starts with the opcode `OP_RETURN 6a`. This is followed by the length bytes `0x20` or 32 bytes. So, there are 32 bytes coming next. The value beginning with `e36c` and its size indicates that it is a hash value of some kind.

We downloaded a app called Notary Block on our iPhone which took a 32-byte hash of a picture. It could be a document, video, sound file; anything. It then added this hash to the Blockchain.

The email we received from the Notary Block app.

```
Document url: https://files.parsetfss.com/4583494d-c199-4b86-9b22-6c4aa84b49eb/tfss-e99b361e-7a0c-4f57-
b5e9-44231839e46f-20160208_1449.jpg
```

```
Date: 2016-02-08
```

```
Description: vijaymukhi
```

```
Hash function: SHA-256
```

Photo hash:
e36c7d4102bb43c8ec3748f23b62c484c5ea7e00c4ea51e6d49743349311f22a
Blockchain: Bitcoin Mainnet
Block No:
Transaction hash:
29b781831f30f1f8d91aba3fb9996000b0c62b4c49aab7d10b6d0bc97754b3b1
Status: Waiting for confirmation.

This app saves the transaction hash value in the Bitcoin blockchain for life. The photo hash in the email is part of the output. This is a transaction output which cannot be spent as the Bitcoin address is not valid.

Again, once something enters the blockchain and is confirmed by at least seven blocks sitting on top of it, then the transaction is cast in stone. So, simply take a hash of something at a certain point in time and it is guaranteed that no one can change its contents.

Let's take an example. A will is made and two people (witnesses) digitally sign it. A sha256 hash value is computed of this electronic document and placed in the blockchain. Now, nobody can dispute the contents of the will as the hash value ensures that the document is not changed. Though, the hash value in blockchains is a public entity and everyone can see it.

Similarly, land records can be digitized. Simply place the hash value in the blockchain. The actual property record can be saved as an image and placed someplace safe. The stored document link in the cloud can also be saved along with the hash value in the output of the transaction and thus in the blockchain. The next buyer of the property can see the entire history of the ownership of the property on the blockchain. It is very clever method of using Bitcoins, especially creating an engine of trust.

Let's now look at another transaction hash.

25adbe994c246e6522b09e58516a902e4b2c4462f052b2abb81c896d5da50d95 which is in block 397946.

The output starts with OP_RETURN, then the words DOCPROOF and thereafter a hash value following it. This company wants the world at large to standardize on the syntax used for interoperability. We had to pay about 130 Indian Rupees, before we could get our hash onto the giant trust machine.

Dust

```
ch1303.py
import bitcoin
import os
mbaddr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
pkey = "L2T9wdbjAvyiXQnm27WPNR323Ce8cDkDkMARQx125CQZ43gHG4yR"
unspend = bitcoin.history(mbaddr)
print "No of transactions %d" % len(unspend)
unspendl = []
i = 0
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)
spendd = unspendl[0]
outs = [{'value': 100 , 'address': '1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv'}]
```

```
tx = bitcoin.mktx(spenddd,outs)
tx1 = bitcoin.sign(tx , 0 , pkey)
os.system("bitcoin-cli sendrawtransaction " + tx1)
```

Output

```
No of transactions 50
error code: -26
error message:
64: dust
```

If the amounts are very small in a transaction, you will get a dust error. The program that creates and signs a Bitcoin transaction from the last chapter now generates an error.

The only change in the program is on the number of Satoshi, it is reduced to 100. The error message with dust simply means, that like a particle of dust, the Bitcoin value is too small. The Bitcoin protocol has a lower limit on the Bitcoins. You cannot send insignificant small values. More on how this value of dust is computed, is left for another day.

CHAPTER 14

Pay to Script Hash or Multi-Sig Bitcoin Addresses

In all the chapters so far, the Bitcoin addresses start with 1. Also, we have had only one private key and this private key generated one public key. Let's now push the envelope a little further, look at bitcoin addresses beginning with 3 and replace the word single/one with multiple. A fair warning here, you will not see the same output on your screen.

Three different Bitcoin addresses are created executing the method `getnewaddress`. This is what we finally see when we execute method

```
$bitcoin-cli getaddressesbyaccount ''
[
  "18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7",
  "1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74",
  "1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7"
]
```

The method `getnewaddress` is executed thrice, therefore there are three different values of Bitcoin addresses displayed on the screen. You will have different addresses from what we have. The blockchain explorer validates them.

Once again, the method `getnewaddress` creates a new Bitcoin address. Now, we require the private key that created this Bitcoin address and the public key associated with this private key.

To access the public key, the following command is executed.

```
$bitcoin-cli validateaddress 1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7
```

Output

```
{
  "isvalid": true,
  "address": "1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7",
  "scriptPubKey": "76a9147eb109268f079ce35848a104ad13938e85716b1b88ac",
  "ismine": true,
  "iswatchonly": false,
  "isscript": false,
  "pubkey": "03fcecc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596",
  "iscompressed": true,
  "account": ""
}
```

The method `validateaddress` returns all kinds of information about the Bitcoin addresses along with the key `pubkey`. Finally, the public key associated with the bitcoin address is obtained. The client program, `bitcoin-cli` with the option `dumpprivkey` is executed to acquire the private key.

```
$bitcoin-cli dumpprivkey 1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7
L2C7ARUHMdHV55pWf9SzCz2Sf11yDYy1iK4M5qcixweJJUT7d3iU
```

In this chapter, we will use the last Bitcoin address, the one starting with 1CY.

Retrieving the Public and Private Key from a Bitcoin Address in the Wallet.

```
ch1401.py
import subprocess
import ast
def keys(baddr):
    raw = subprocess.check_output(["bitcoin-cli", "validateaddress", baddr])
    raw = raw.replace('true', '"true"')
    raw = raw.replace('false', '"false"')
    outdict = ast.literal_eval(raw)
    pub = outdict['pubkey']
    raw = subprocess.check_output(["bitcoin-cli", "dumpprivkey", baddr])
    pkey = raw[:-1]
    return(pub, pkey)

baddr1 = '18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7'
baddr2 = '1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74'
baddr3 = '1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7'

(pub1, pkey1) = keys(baddr1)
(pub2, pkey2) = keys(baddr2)
(pub3, pkey3) = keys(baddr3)

print baddr1, pub1, pkey1
print baddr2, pub2, pkey2
print baddr3, pub3, pkey3
```

Output

```
18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7
028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b
KyH6Mm8sYmnT9cm2jMmxpae7CAvbCmmxQ1bNysjAeMeeT2XD6KWp
1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74
035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee
L2zUVV86B3Lxf9PLCzmzQjTbLUSWsPjvAGFVvyR7CKUMavxSW7c
1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7
03fcecc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596
L2C7ARUHMdHV55pWf9SzCz2Sf11yDYy1iK4M5qcixweJJUT7d3iU
```

This program has a function called `keys` which executes the previous two methods with the client program, `bitcoin-cli` and returns their output. In the function, the values returned by the `validateaddress` option cannot be converted into a Python dictionary in its current form because some keys in the string are of boolean type. So, both the values, `true` and `false` are converted from boolean objects into strings.

Then the `literal_eval` function returns the string as a dictionary, saved in `outdict`. The public key is accessed using the key called `pubkey`.

Following this, the method `dumpprivkey` is used to return the private key, the hidden enter at the end of the string must be eliminated. A tuple of public key and private key is returned.

Caution. Please make sure that the Bitcoin address used in your program are the ones you get otherwise you will see errors/exceptions thrown with our code.

Just warming up, nothing new.

Creating a Bitcoin Address Starting with a 3, a Multi-Sig Address

```
ch1402.py
import subprocess
import ast
def keys(baddr):
    raw = subprocess.check_output(["bitcoin-cli" , "validateaddress" , baddr ])
    raw = raw.replace('true' , "'true'")
    raw = raw.replace('false' , "'false'")
    outdict = ast.literal_eval(raw)
    pub = outdict['pubkey']
    raw = subprocess.check_output(["bitcoin-cli" , 'dumpprivkey' , baddr])
    pkey = raw[:-1]
    return(pub ,pkey )

baddr1 = '18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7'
baddr2 = '1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74'
baddr3 = '1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7'

(pub1 , pkey1) = keys(baddr1)
(pub2 , pkey2) = keys(baddr2)
(pub3 , pkey3) = keys(baddr3)

str0 = ["%s" , "%s" , "%s"] % (pub2 , pub1 , pub3)
raw = subprocess.check_output(["bitcoin-cli" , "createmultisig" , '2' , str0 ])
outdict = ast.literal_eval(raw)
fbaddr = outdict['address']
rScript = outdict['redeemScript']
print raw
print "Final MultiSig Bitcoin address %s" % fbaddr
print "RedeemScript %s" % rScript
```

Output

```
{
    "address": "3KomW7R867cfsyQ7wg3pkC358E9j57PK3M",
    "redeemScript":
    "5221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee21
    028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fce
    cc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae"
}
Final MultiSig Bitcoin address 3KomW7R867cfsyQ7wg3pkC358E9j57PK3M
RedeemScript
```

```
5221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee210
28003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fcec
c0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae
```

The program focusses on something very different, very new. A Bitcoin address associated with multiple public keys is created.

The method called `createmultisig` is the taskmaster here. This method takes two parameters. The first parameter is a count, as in the number of private keys required to unlock the Bitcoins present in this Bitcoin address. The number 2 implies that a minimum of 2 private keys associated with the public keys is mandatory for permitting the transfer of ownership of Bitcoins. There will be many public keys in the list, but only some of them will be used.

The second parameter is a list/array containing all the public keys in a string format that can be used indirectly in unlocking the Bitcoins. In our case, 3 public keys associated with the 3 Bitcoin addresses are given. These public keys undoubtedly, will be very different from yours.

The end-result is a Bitcoin address that starts with a 3 and not a 1 like all the Bitcoin addresses we have seen so far.

Any Bitcoin addresses that starts with 3 requires signatures from more than one private key to unlock the Bitcoins trapped in them. In this specific case, minimum 2 private keys associated with the 3 public keys are needed to unlock the Bitcoins stored. The basic principle here is that, 2 out of 3 people who hold the private keys, must sign the transaction. Only after this condition is met can the Bitcoins be transferred.

The option `createmultisig` returns a dictionary with only two keys, namely `address` and `redeemScript`. We will deal with the `redeemScript` key in the next program. The variable `fbaddr` now has the Bitcoin address starting with 3. It is assumed that some Bitcoins are there in this multisig address so that its ownership can be transferred to another Bitcoin address, starting with 1. So, transfer some Bitcoins to the Bitcoin address beginning with 3Kom. Once done, enter the above Bitcoin address in the blockchain explorer, you will see over seven transactions with it.

Remember, Bitcoin money transfer is a slow process as it takes some time before the transactions gets confirmed. Our plan is to transfer Bitcoins from this Bitcoin address to our old Zebpay Bitcoin address starting with 1KF.

To sum up, a Bitcoin address starting with 3 is different from a Bitcoin address starting with 1. Bitcoin addresses starting with 1 depend upon one private key. On the other hand, Bitcoin addresses starting with 3 get their address values from the specified public keys, which in turn depend upon private keys. These addresses are not random. Nearly all code on the web uses a Bitcoin address using only 3 public keys.

Decoding the RedeemScript Key of a Multi Signature Bitcoin Address

```
ch1403.py
import subprocess
import ast
scriptd = {
    0x51: 'OP_1',
    0x52: 'OP_2',
    0x53: 'OP_3',
    0x54: 'OP_4',
    0x55: 'OP_5',
    0x56: 'OP_6',
    0x57: 'OP_7',
    0x58: 'OP_8',
    0x59: 'OP_9',
```

```
    0x60:'OP_10',
    0x61:'OP_11',
    0x62:'OP_12',
    0x63:'OP_13',
    0x64:'OP_14',
    0x65:'OP_15',
    0x66:'OP_16',
    0xae:'OP_CHECKMULTISIG'
}

def redeem(script):
    script = script.decode('hex')
    i = 0
    ans = ''
    while i < len(script):
        byte = ord(script[i])
        if byte >= 0x51 and byte <= 0x66:
            ans = ans + scriptd[byte] + " "
        elif byte == 0xae:
            ans = ans + scriptd[byte]
            i = i + 1
        else:
            length = ord(script[i])
            i = i + 1
            pubkey = script[i:i + length]
            ans = ans + pubkey.encode('hex') + " "
            i = i + length - 1
            i = i + 1
    return ans

def keys(baddr):
    raw = subprocess.check_output(["bitcoin-cli", "validateaddress", baddr])
    raw = raw.replace('true', '"true"')
    raw = raw.replace('false', '"false"')
    outdict = ast.literal_eval(raw)
    pub = outdict['pubkey']
    raw = subprocess.check_output(['bitcoin-cli', 'dumpprivkey', baddr])
    pkey = raw[:-1]
    return(pub, pkey)

baddr1 = '18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7'
baddr2 = '1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74'
baddr3 = '1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7'

(pub1, pkey1) = keys(baddr1)
(pub2, pkey2) = keys(baddr2)
(pub3, pkey3) = keys(baddr3)

str0 = "[%s", "%s", "%s]" % (pub2, pub1, pub3)
```



```

raw = subprocess.check_output(["bitcoin-cli", "createmultisig", '2', str0 ])
outdict = ast.literal_eval(raw)
fbaddr = outdict['address']
rScript = outdict['redeemScript']
print "Final MultiSig Bitcoin address %s" % fbaddr
print "RedeemScript %s" % rScript

ans = redeem(rScript)
print ans

```

Output

```

Final MultiSig Bitcoin address 3KomW7R867cfsyQ7wg3pkC358E9j57PK3M
RedeemScript
5221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee210
28003e 21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fce
cc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae
OP_2 035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee
028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b
03fcec0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596 OP_3
OP_CHECKMULTISIG

```

In this program, we are only deciphering the value of the redeemScript key. The Bitcoin programmers have written the code to generate this value. The two elements used in code are again the number of private keys required to unlock the data and a list of public keys with their values. The redeemScript key has this information.

The function called redeem is given this script and it returns a decoded string or an English readable redeemScript. The earlier program had a bunch of hex numbers that meant nothing to nobody.

The Bitcoin script engine has pre-defined opcodes for the numbers 1 to 16. Here, OP_1 or hex bytes 0x51 stands for the number 1. The number 2 or opcode OP_2 is hex byte 0x52 and so on and so forth. Thus, opcode OP_1 to OP_16 are represented by hex bytes ranging from 0x51 to 0x66.

The only task of the redeem function is to interpret this redeemScript value.

In the function, first the hex bytes passed in the script parameter are decoded. Then the first byte is extracted as a number using the ord function. A check is performed for the byte to be in the range of 0x51 and 0x66. If yes, then it is the byte that determines the number of public keys to be used and more for signing the transaction.

The first byte is the number of keys required, it is followed by the total number of public keys. Its value is 0x52, thus implying that only two keys are required to sign off on the Bitcoin address. The createmultisig function in the earlier program also used two keys.

Then comes the public keys, prefaced with the length bytes. The length of public key is not constant, it changes depending on the type of public key, whether it is compressed or not. A ECDSA compressed public key key is 33 bytes large. In the earlier days of Bitcoin, public keys were uncompressed and 65 bytes large.

```

52
21
035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee
21
028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b
21

```

```
03fcecc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596
53
ae
```

The redeem script has been broken up to clearly see the three public keys. These public keys start with the length bytes. In the earlier program, the redeem script used the same public keys.

The public keys are placed one after the other. The first byte or OP_2 also comes from the first parameter of the createmultisig method.

Coming back to our program. In the else block, the length of the public key is read and then the index variable i is incremented by 1 because only one byte is read. No one knows how many public keys follow. The first byte of the redeem script key is for the number of public/private keys required and not the total number of public keys in the script.

The slice operator is used to read in the public key. Then the index variable i is increased to point to the length byte of the next public key. You will notice that the index variable i is increased by 1 at the end of the loop and one less in the else statement.

This process is repeated until a number from 0x51 to 0x66 is encountered. This number, n represents the total number of public keys encountered so far. A minimum number, m is chosen from this number, n. m is smaller than n.

This number is the second last byte of the redeem script and it comes from the length of the list passed earlier.

Finally, we come to the end, the last byte is 0xae. This byte stands for the opcode OP_CHECKMULTISIG or in plain English, signatures that is in plural or many signatures following. We could have also read the second last byte and determined the total number of public keys. This assumes the last byte is the opcode 0xae.

The opcode OP_CHECKMULTISIG is handled differently.

Creating a Multi-Signature Bitcoin Address using a Python Library Instead

```
ch1404.py
import subprocess
import ast
import pybitcointools
def keys(baddr):
    raw = subprocess.check_output(["bitcoin-cli", "validateaddress", baddr])
    raw = raw.replace('true', '"true"')
    raw = raw.replace('false', '"false"')
    outdict = ast.literal_eval(raw)
    pub = outdict['pubkey']
    raw = subprocess.check_output(['bitcoin-cli', 'dumpprivkey', baddr])
    pkey = raw[:-1]
    return(pub, pkey)

baddr1 = '18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7'
baddr2 = '1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74'
baddr3 = '1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7'

(pub1, pkey1) = keys(baddr1)
(pub2, pkey2) = keys(baddr2)
(pub3, pkey3) = keys(baddr3)
```

```

str0 = ["%s" , "%s" , "%s"] % (pub2 , pub1 , pub3)
raw = subprocess.check_output(["bitcoin-cli" , "createmultisig" , '2' , str0 ])
outdict = ast.literal_eval(raw)
fbaddr = outdict['address']
rScript = outdict['redeemScript']
print "Final MultiSig Bitcoin address %s" % fbaddr
print "RedeemScript %s" % rScript

pkeyl = []

pkeyl.append(pub2)
pkeyl.append(pub1)
pkeyl.append(pub3)

raws = pybitcointools.mk_multisig_script( pkeyl , 2)
print raws
script = pybitcointools.deserialize_script(raws)
print script
baddr = pybitcointools.scriptaddr(raws)
print "PyBitcoins Bitcoins Address is %s" % baddr
if baddr == '3KomW7R867cfsyQ7wg3pkC358E9j57PK3M':
    print "OK"
else:
    print "Vijay Mukhi is an embecile"

```

Output

```

Final MultiSig Bitcoin address 3KomW7R867cfsyQ7wg3pkC358E9j57PK3M
RedeemScript
5221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee210
28003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fcec
c0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae
5221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee210
28003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fcec
c0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae
[2, '035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee',
'028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b',
'03fcec0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596', 3,
174]
PyBitcoins Bitcoins Address is 3KomW7R867cfsyQ7wg3pkC358E9j57PK3M
OK

```

A multi-sig Bitcoin address beginning with 3 can be created using the pybitcointools module also. It offers the same functionality. So, it's entirely up to the coder, either to use the original official code from the Bitcoin core codebase or functions from a Python library.

The function, mk_multisig_script generates the same redeemScript. The raws variable is assigned this value. Two parameters are given as before, the list of public keys and the number of private keys needed. The only difference is that the order is reversed. The placement is very important, change this order and the Bitcoin address also changes.

Then there is a function called deserialize_script that decodes the script and returns a list, which is a lot easier to read.

This list starts with the number of private keys needed to sign, followed by the three public keys, again followed by the total number of public keys that have been specified and finally the opcode OP_CHECKMULTISIG. The number 174 in decimal is 0xae in hex, the opcode of OP_CHECKMULTISIG.

Finally, the original script is passed to the function `scriptaddr` which returns a multi-sig Bitcoin address. As there are many changes constantly being made to the code, there is an if statement to check if the Bitcoin addresses match. It does J.

The moral of this story, use either a library like `pybitcointools/bitcoin` or code from `bitcoin-cli`, the Bitcoin address generated will be the same.

Creating a Multi-Sig Bitcoin Address One Line at a Time or by Hand

```
ch1405.py
import subprocess
import ast
import pybitcointools
from cfuncs import *
def keys(baddr):
    raw = subprocess.check_output(["bitcoin-cli", "validateaddress", baddr ])
    raw = raw.replace('true', '"true"')
    raw = raw.replace('false', '"false"')
    outdict = ast.literal_eval(raw)
    pub = outdict['pubkey']
    raw = subprocess.check_output(["bitcoin-cli", "dumpprivkey", baddr])
    pkey = raw[:-1]
    return(pub ,pkey )

baddr1 = '18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7'
baddr2 = '1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74'
baddr3 = '1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7'

(pub1 , pkey1) = keys(baddr1)
(pub2 , pkey2) = keys(baddr2)
(pub3 , pkey3) = keys(baddr3)

str0 = "[" + "%s" , "%s" , "%s" % (pub2 , pub1 , pub3)
raw = subprocess.check_output(["bitcoin-cli", "createmultisig", '2' , str0 ])
outdict = ast.literal_eval(raw)
fbaddr = outdict['address']
rScript = outdict['redeemScript']
print "Final MultiSig Bitcoin address %s" % fbaddr
print "RedeemScript %s" % rScript

pkeyl = []

pkeyl.append(pub2)
pkeyl.append(pub1)
pkeyl.append(pub3)

dict = {1:"51" , 2:"52" , 3:"53" , 174: "ae" }
nopkey = 2
```

```

tpkey = 3
opcode = 174
length = len(pub1)/2
lenpublickey = "%x" % length
rawscript = dict[nopkey] + lenpublickey + pub2 + lenpublickey + pub1 + \
    lenpublickey + pub3 + dict[tpkey] + dict[opcode]
print rawscript
rawscript = rawscript.decode('hex')
intermed = hashlib.sha256(rawscript).digest()
digest = hashlib.new('ripemd160', intermed).digest()
inp_fmt = chr(int(5)) + digest
checksum = chash(inp_fmt)[4:]
baddr = pybitcointools.changebase(inp_fmt+checksum, 256, 58)
print "PyBitcoins Bitcoin Address is %s" % baddr
if baddr == '3KomW7R867cfsyQ7wg3pkC358E9j57PK3M':
    print "OK"
else:
    print "Vijay Mukhi is an embecile"

```

Output

```

Final MultiSig Bitcoin address 3KomW7R867cfsyQ7wg3pkC358E9j57PK3M
RedeemScript
5221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee210
28003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fcec
c0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae
5221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee210
28003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fcec
c0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae
PyBitcoins Bitcoin Address is 3KomW7R867cfsyQ7wg3pkC358E9j57PK3M
OK

```

This example shows the same result as the previous one but here our focus is on something very basic. How do the pybitcointools functions compute Bitcoin addresses?

We will create a multi-sig Bitcoin address step by step.

First, a small dictionary object called dict is created which has only those opcodes that are noticeable in a standard multi signature redeemScript. The redeemScript variable rawscript is coded by hand since we know the format. It is a list and it starts with the number of private keys required, not the number 2 but the opcode 0x52. Then is the length of individual public key and following it is the actual public key, the same is repeated for all the public keys. These public keys are stored sequentially. In our case, this process is repeated thrice. At the end, there is the opcode for 3 public keys OP_3 and finally the opcode for a multi signature 0xae.

The loops are avoided as these hex values are concatenated together using variables. Lastly, the encoded string variable rawscript is decoded.

All Bitcoin addresses must be a Sha256 hash value followed by a RipeMD hash value, which is converted to a Bitcoin address using base58 encoding. Earlier the sha hash value of the public key was computed but here, the sha hash value of the redeem script is taken. In this manner, all the public keys are hashed together.

After the RipeMD hash value is computed, A number is inserted at the start to this 160-bit hash. Earlier a 0 was added at the start of the ripemd hash to get Bitcoin addresses starting with a 1. Now a 5 is added at the beginning so that the Bitcoin address starts with a 3.

All this is learnt from reading and copying the source code of the module pybitcointools.

The same code used for calculating a normal Bitcoin address is used here with two minor changes. Finally, the checksum of the RipeMD hash is calculated of which only the first 4 bytes are taken and then the sha hash is calculated using the same chash function. This is stored in checksum variable.

Next is the base58 encoding. Everything else remains the same, the only difference is in the redeem script and the number 5. All this is messy, but boosts our confidence since we have been able to successfully create a multi-sig Bitcoin address.

Sending Bitcoins from a Multi-Sig Address to a Normal Bitcoin Address

```
ch1406.py
import subprocess
import ast
import bitcoin
from cfuncs1d import *
import os
def keys(baddr):
    raw = subprocess.check_output(["bitcoin-cli", "validateaddress", baddr ])
    raw = raw.replace('true', '"true"')
    raw = raw.replace('false', '"false"')
    outdict = ast.literal_eval(raw)
    pub = outdict['pubkey']
    raw = subprocess.check_output(["bitcoin-cli", 'dumpprivkey', baddr])
    pkey = raw[:-1]
    return(pub ,pkey )

baddr1 = '18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7'
baddr2 = '1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74'
baddr3 = '1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7'

(pub1 , pkey1) = keys(baddr1)
(pub2 , pkey2) = keys(baddr2)
(pub3 , pkey3) = keys(baddr3)

str0 = "[" + "%s" , "%s" , "%s" ] % (pub2 , pub1 , pub3)
raw = subprocess.check_output(["bitcoin-cli", "createmultisig", '2' , str0 ])
outdict = ast.literal_eval(raw)
fbaddr = outdict['address']
rScript = outdict['redeemScript']
print "Final MultiSig Bitcoin address %s" % fbaddr
print "RedeemScript %s" % rScript
unspend = bitcoin.history(fbaddr)
unspendl = []
i = 0
for obj in unspend:
```

```

        if 'spend' not in obj.keys():
            unspendl.append(obj)

spendd = unspendl[0]
print spendd
newobject = spendd['output']
ind = newobject.index(':')
thash = newobject[0:ind]
outi = int(newobject[ind + 1:])
print "Transaction hash %s" % thash
print "outi %d" % outi

scr = scrpubkey(thash , outi)
print "Script Public key %s" % scr

dbaddr = "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
bitcoins = 0.000400
str1 = '[{"txid": "%s", "vout": %d, "scriptPubKey": "%s", "redeemScript": "%s"}]' % \
    (thash , outi , scr, rScript)
str2 = '{"%s": %f}' % (dbaddr , bitcoins)
tbytes = subprocess.check_output(["bitcoin-cli" , "createrawtransaction" , str1 , str2])
tbytes = tbytes[:-1]
print "-----Transaction Bytes minus the Signature"
os.system("bitcoin-cli decoderawtransaction " + tbytes)
str3 = '[{"txid": "%s", "vout": %d, "scriptPubKey": "%s", "redeemScript": "%s"}]' % \
    (thash , outi, scr, rScript)
str4 = '["%s"]' % pkey1
raw = subprocess.check_output(["bitcoin-cli" , "signrawtransaction" , tbytes , str3 , str4])
raw = raw.replace('true' , "true")
raw = raw.replace('false' , "false")
outdict = ast.literal_eval(raw)
fbytes = outdict['hex']
print "We have an error message which we ignore"
print raw
print "-----Transaction Bytes with only one Signature"
os.system("bitcoin-cli decoderawtransaction " + fbytes)
str5 = '[{"txid": "%s" , "vout": %d, "scriptPubKey": "%s" , "redeemScript": "%s"}]' \
    % (thash , outi , scr , rScript)
str6 = '["%s"]' % pkey2
raw = subprocess.check_output(["bitcoin-cli" , "signrawtransaction" , fbytes , str5 , str6 , "ALL"])
raw = raw[:-1]
raw = raw.replace('true' , "true")
raw = raw.replace('false' , "false")
outdict = ast.literal_eval(raw)
rawtbytes = outdict['hex']
print raw
print "Transaction Bytes with 2 signatures"
os.system("bitcoin-cli decoderawtransaction " + rawtbytes)

```

```
raw = subprocess.check_output(["bitcoin-cli", "sendrawtransaction", rawtbytes])
print raw
```

Output

```
Final MultiSig Bitcoin address 3KomW7R867cfsyQ7wg3pkC358E9j57PK3M
RedeemScript
5221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee210
28003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fcec
c0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae
{'output':
  u'f00fcb545832cca766dbe8c07a91db637636f31ac17b33aae4aa66ed17dd8b03:1',
  'block_height': 418062, 'value': 59800, 'address':
  u'3KomW7R867cfsyQ7wg3pkC358E9j57PK3M'}
Transaction hash
f00fcb545832cca766dbe8c07a91db637636f31ac17b33aae4aa66ed17dd8b03
outi 1
Script Public key a914c6b9233917283123627a222e8c6826d61c84d67587
-----Transaction Bytes minus the Signature
{
  "txid": "134b0db22e1541bc0e81eab90046c02528de9b46f42f76249b77437cacaca958",
  "size": 85,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "f00fcb545832cca766dbe8c07a91db637636f31ac17b33aae4aa66ed17dd8b03",
      "vout": 1,
      "scriptSig": {
        "asm": "",
        "hex": ""
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00040000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
        OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
        ]
      }
    }
  ]
}
```



```

    }
  }
]
}
We have an error message which we ignore
{
  "hex":
    "010000001038bdd17ed66aae4aa337bc11af3367663db917ac0e8db66a7cc325854cb0
    ff001000000b40047304402203bf3e72d49f3347b7551c52feec010b2cc87ee152cd72f7e
    f9a80c97211473f402200746f77f35c71c9933fe8eea5cf54bcada6d37adef81e5a07194c
    1305a73bf41014c695221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6
    522b0e80e789ee21028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4
    707e05b36b2103fccc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d566
    2c59653aefffffffff01409c0000000000001976a914c847b6d84ecf8048474b19c4a233040
    9ff32a1ad88ac000000000",
  "complete": "false",
  "errors": [
    {
      "txid":
        "f00fcb545832cca766dbe8c07a91db637636f31ac17b33aae4aa66ed17dd8b03",
      "vout": 1,
      "scriptSig":
        "0047304402203bf3e72d49f3347b7551c52feec010b2cc87ee152cd72f7ef9a80c972114
        73f402200746f77f35c71c9933fe8eea5cf54bcada6d37adef81e5a07194c1305a73bf410
        14c695221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789
        ee21028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b21
        03fccc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae",
      "sequence": 4294967295,
      "error": "Operation not valid with the current stack size"
    }
  ]
}

```

-----Transaction Bytes with only one Signature

```

{
  "txid": "f305a8dc77566233077035f29e1de470f4e5c3e3fdef90bad17c7ebd44ada253",
  "size": 265,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid":
        "f00fcb545832cca766dbe8c07a91db637636f31ac17b33aae4aa66ed17dd8b03",
      "vout": 1,
      "scriptSig": {
        "asm": "0
        304402203bf3e72d49f3347b7551c52feec010b2cc87ee152cd72f7ef9a80c97211473f40

```

```
2200746f77f35c71c9933fe8eea5cf54bcada6d37adef81e5a07194c1305a73bf41[ALL]
5221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee210
28003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fcec
c0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae",
"hex":
"0047304402203bf3e72d49f3347b7551c52feec010b2cc87ee152cd72f7ef9a80c972114
73f402200746f77f35c71c9933fe8eea5cf54bcada6d37adef81e5a07194c1305a73bf410
14c695221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789
ee21028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b21
03fcec0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae"
},
"sequence": 4294967295
}
],
"vout": [
{
"value": 0.00040000,
"n": 0,
"scriptPubKey": {
"asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
OP_EQUALVERIFY OP_CHECKSIG",
"hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
"reqSigs": 1,
"type": "pubkeyhash",
"addresses": [
"1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
]
}
}
]
}
{
"hex":
"0100000001038bdd17ed66aae4aa337bc11af3367663db917ac0e8db66a7cc325854cb0
ff001000000fd00004830450221008f789cd6eefc7da2e7106014ddd2f6953861e9278b
17e91ef8356ede02306b94022012081b697cfe119bc07a66839347c1faf4f2db5ae680a1
04256b7f404ea810400147304402203bf3e72d49f3347b7551c52feec010b2cc87ee152c
d72f7ef9a80c97211473f402200746f77f35c71c9933fe8eea5cf54bcada6d37adef81e5a0
7194c1305a73bf41014c695221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198ef
af3d6522b0e80e789ee21028003e21c092ba3facaef44f92b3063389d345d03f54ac14933
f7b4707e05b36b2103fcec0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb
3d5662c59653aeffffffff01409c0000000000001976a914c847b6d84ecf8048474b19c4a2
330409ff32a1ad88ac00000000",
"complete": "true"
}
}
Transaction Bytes with 2 signatures
```

```

{
  "txid":
    "2616c318f15ff1e696f71c253d0ccea615b4e0ebb4c9631ed5e4de155262c3db",
  "size": 340,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid":
        "f00fcb545832cca766dbe8c07a91db637636f31ac17b33aae4aa66ed17dd8b03",
      "vout": 1,
      "scriptSig": {
        "asm": "0
30450221008f789cd6eefc7da2e7106014ddd2f6953861e9278b17e91ef8356ede02306b
94022012081b697cfe119bc07a66839347c1faf4f2db5ae680a104256b7f404ea81040[AL
L]
304402203bf3e72d49f3347b7551c52feec010b2cc87ee152cd72f7ef9a80c97211473f40
2200746f77f35c71c9933fe8eea5cf54bcada6d37adef81e5a07194c1305a73bf41[ALL]
5221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee210
28003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fcec
c0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae",
        "hex":
          "004830450221008f789cd6eefc7da2e7106014ddd2f6953861e9278b17e91ef8356ede0
2306b94022012081b697cfe119bc07a66839347c1faf4f2db5ae680a104256b7f404ea810
400147304402203bf3e72d49f3347b7551c52feec010b2cc87ee152cd72f7ef9a80c97211
473f402200746f77f35c71c9933fe8eea5cf54bcada6d37adef81e5a07194c1305a73bf41
014c695221035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e78
9ee21028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2
103fcccc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00040000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
        ]
      }
    }
  ]
}

```

```
    }  
  }  
]  
}  
2616c318f15ff1e696f71c253d0ccea615b4e0ebb4c9631ed5e4de155262c3db
```

This program is very, very, long but it has successfully sent across a transaction which unlocks a multi-sig Bitcoin addresses.

To confirm what we have done is acceptable to the Bitcoin miners, the transaction hash value starting with 2616 on the last line of the output is keyed into our favorite blockchain explorer. Our multi-sig bitcoin address starting with 3kom is visible in the upper left corner in the inputs section.

So, let's understand the program. Like before, the history function is used to return all the outputs that can be spent by the Bitcoin address starting with 3Kom. The code to create this Bitcoin address and the redeem script remains the same.

The focus here is on the transaction hash and the output index of the output within that transaction. For the absent-minded, the transaction hash value and the output index will be seen in the input of the new transaction. The new transaction is the one where we are the owners and these Bitcoins will be sent to the address starting with 1KFz.

Looking at the output, the transaction hash starts with f00b. The blockchain explorer shows that the second Bitcoin address starts with 3Kom. Thus, it indicates that the Bitcoins were transferred to the multi-sig address in June 2016. Time flies when you are working.

Let's now focus on the scriptPublicKey. Earlier, Bitcoins were sent to an address starting with 1. These Bitcoin addresses following a certain set of rules.

The basic principle of multi-sig Bitcoin addresses is that whoever unlocks these Bitcoins will require two private keys. Otherwise, you are out of luck. The sender of Bitcoins remains clueless of this requirement. For the sender, the interaction with this address or any address remains the same. Nothing changes.

When a transaction with a Bitcoin address must be signed twice, both parties need a valid private key that creates the three public keys. The logic here is to distribute the three private keys to three separate people and any two of them can sign the transaction to release the funds. Most corporates follow this process where a cheque must be signed by at least say two people.

The idea is to claim ownership of these Bitcoins, one private key is not enough. More than one private key is required for the transfer of ownership to take place. This other person could be a trustee chosen by two competing parties. Only if the trustee along with the other party approves the transaction, the Bitcoins will be transferred. Complex smart contracts are built with this logic, using the Pay to Script Hash or P2SH.

Earlier the Bitcoin address was a simple random number, now it stands for a script which has four pieces of information. First is the total number of private keys required to sign the transaction, next is the total number of public keys, third is the actual public keys and finally, the opcode for a multiple signature. We use the word private and public interchangeably.

Now, the inputs to a transaction carry code. Earlier in the input script field, there was a public key that hashed to a Bitcoin address starting with 1 along with the signature. Now there is a redeem script which hashes to a Bitcoin address starting with a 3.

All that's happened is that a single key in the input is replaced with a more complex redeemScript key. The useless Bitcoin address now represents an actual script and public keys and more.

In the program, a variable `dbaddr` is created to hold the Bitcoin address `1KFz`. Once again, the Bitcoins will go to the same Bitcoin address we have been using so far. We advise you to change this address to yours.

A gentle reminder. We are sending across a very small value, i.e. `000400`. This is because the higher the miner fees, the faster the miners accept this transaction. If the fee is low then the transaction will simply not enter the blockchain. It will become history and dust.

We could have used the testnet network to accomplish the same thing but working with a live network is real Monty. With Bitcoin's testnet network, you do not have to pay real money to buy Bitcoins. You become a miner, the Bitcoins you mine are less valuable than junk.

Let's understand the output. Bitcoins are fetched from a hash value that starts with `f00f` and it is the first and not 0th output. The value of variable `outi` is 1. Please look at the block explorer's output for this transaction hash value. It starts with the same `OP_HASH160` opcode, followed by the hash. But it ends with an opcode `OP_EQUAL` which is every different from the earlier script opcodes. The public key like before, is placed in a variable called `scr` using the same function `scripubkey`.

A new transaction is created with the same method `createrawtransaction` and the transaction hash starts with `134b`. You will see a very different value. This is the transaction id of our transaction on the Bitcoin network. This is a however a very temporary hash as the signature bytes have not yet been placed in the input of the transaction. The field `scriptSig` is empty.

So, the strategy is to create an unfinished transaction using the method `createrawtransaction` and give it the inputs and outputs as two separate strings. For the inputs, the hash value of the transaction with the unspent Bitcoins is given and the output index in the above transaction. Then the `scriptPublicKey` stored in the variable `scr` is passed. The `redeemScript` is the only new addition, most of it is the same.

The output is however much simpler, there are only two values, the destination Bitcoin address which starts with a 1 and the Bitcoin value.

Let's look at the actual transaction this method creates.

The transaction hash or `txid` is a computed value. This value keeps changing till the transaction is signed so it can be ignored for the time being.

The `txid` field of the list `vin` is of the transaction hash value that contains the multi-sig Bitcoin address. The hash value starts with `f00f`. As the transaction is not signed, the field `scriptSig` is empty. This value is given to the `createrawtransaction` method.

In the `vout` or outputs section, everything looks the same. The type key of the output confirms it is a simple `pubkeyhash` output.

To sign this transaction, the only known method, `signrawtransaction` is used. The raw transaction bytes returned by the `createrawtransaction` is supplied to it after deleting the last enter mark. The variables `str1` and `str3` contain the same data. While creating the transaction, the transaction hash value and the output index are needed. However, when signing it, this data is not required. It's already placed in the transaction inputs. In the same vein, the `scriptPublicKey` and `redeem script` is required only while signing and not creating the transaction.

One private key to sign the transaction is passed as the fourth parameter. This gives an error message for no apparent reason, so it is ignored.

The key complete is smart enough to know that only two keys are required for signing and not three. Therefore, the complete key has a value of false. The transaction is signed only once.

Fortunately, the raw transaction bytes are available and can be decoded. The `txid` field is different from the earlier one

as some more data is added in the scriptSig field. Everything else remains the same. For the record, the transaction hash value now begins with f305.

```
00
47
304402203bf3e72d49f3347b7551c52feec010b2cc87ee152cd72f7ef9a80c97211473f40
2200746f77f35c71c9933fe8eea5cf54bcada6d37adef81e5a07194c1305a73bf4101
4c
69
52
21
035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee
21
028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b
21
03fcecc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596
53
ae
```

The displayed bytes are the hex values in the scriptSig field. Due to a bug in the code, yes, we see a false or an opcode OP_FALSE at the start which is a simple 00. Then comes a signature. This starts with the length of the signature, which is 0x47 and then the signature bytes which start with a 0x30. The signature bytes keep changing but it is still a valid signature.

The script does not permit placing any bytes on the stack directly. The opcode 97 or 0x61 is the opcode for a no op or do nothing. All opcodes less than 75 are opcodes that push several bytes on the stack.

To push 200 bytes on the stack, the opcode 0x4c or decimal 76 is used, this opcode is called OP_PUSHDAT1. This opcode looks at the next byte to decide how many bytes need to be pushed on the stack. But if the number of bytes is larger than 255, then the opcode 0x4d or OP_PUSHDAT2 is used, the next two bytes decide the value. These two bytes give a short value.

In this case, 0x69 hex bytes are placed on the stack. It is the same redeem script we worked with earlier. It starts with an OP_2, three public keys and ends with OP_3 and then OP_CHECKMULTISIG. After the signature in this case, comes the redeem script which carries the rules of the game, how many signatures to be used. This is called using m out of n signatures.

As the redeemScript carries public keys and not private keys, these public keys can be used later to verify the signatures signed by the private keys. The redeem script is on the stack, so the public keys can be used to check if it hashes to the right Bitcoin addresses.

Now to signing with the second private key. This value is stored in variable str6. The variables str1, str3 and str5 are the same. Only the private key changes. Only two signatures required. The only change is that the bytes passed to the method change is from the variable, fbytes and not tbytes.

```
00
48
30450221008f789cd6eefc7da2e7106014ddd2f6953861e9278b17e91ef8356ede02306b
94022012081b697cfe119bc07a66839347c1faf4f2db5ae680a104256b7f404ea8104001
47
304402203bf3e72d49f3347b7551c52feec010b2cc87ee152cd72f7ef9a80c97211473f40
```

```

2200746f77f35c71c9933fe8eea5cf54bcada6d37adef81e5a07194c1305a73bf4101
4c
69
52
21
03fcecc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596
21
028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b
21
035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee
53
ae

```

The output shown above is the result after signing the bytes twice. To start with, there is a 00 or false. Then, the 0x48 means the next 144 digits of the first signature are to be read. The following 0x47 is the size of the second signature. So, the 142 digits of the second signature are read. Remember, there are two signatures as its been signed twice.

The opcode 0x4c indicates that the next byte following is the length of bytes to be pushed on the stack. Everything now remains the same as before, except that we end up with two signatures and not one.

No errors reported as the complete key has a value of true. The transaction is now signed and sealed, it is the final transaction ready for dispatch. The txid hash is 2616, which is the same value displayed at the end of the code.

Finally, these raw bytes are sent across using the sendrawtransaction method. The displayed transaction id matches the actual transaction.

Our first multi sig transaction accepted by the miners. J

Creating a Multi-Sig Bitcoin Address that Needs all Three Private Keys to Sign

```

ch1407.py
import subprocess
import ast
def keys(baddr):
    raw = subprocess.check_output(["bitcoin-cli", "validateaddress", baddr ])
    raw = raw.replace('true', '"true"')
    raw = raw.replace('false', '"false"')
    outdict = ast.literal_eval(raw)
    pub = outdict['pubkey']
    raw = subprocess.check_output(['bitcoin-cli', 'dumpprivkey', baddr])
    pkey = raw[:-1]
    return(pub ,pkey )

baddr1 = '18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7'
baddr2 = '1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74'
baddr3 = '1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7'

(pub1 , pkey1) = keys(baddr1)
(pub2 , pkey2) = keys(baddr2)
(pub3 , pkey3) = keys(baddr3)

str0 = "[%s", "%s", "%s"] % (pub2 , pub1 , pub3)

```

```
raw = subprocess.check_output(["bitcoin-cli", "createmultisig", '3', str0 ])
outdict = ast.literal_eval(raw)
fbaddr = outdict['address']
rScript = outdict['redeemScript']
print raw
print "Final MultiSig Bitcoin address %s" % fbaddr
print "RedeemScript %s" % rScript
{
    "address": "38aLpSoCRdDWzj8W5cbS9JzD5Av2qWB8Sj",
    "redeemScript":
    "5321035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee21
    028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fcec
    c0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae"
}
```

Output

```
Final MultiSig Bitcoin address 38aLpSoCRdDWzj8W5cbS9JzD5Av2qWB8Sj
RedeemScript
5321035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee210
28003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fcec
c0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae
```

There is a small change in our code, the earlier program used two of the three private keys to sign the transaction but here, all three private key holders must sign or else it's a no go. This small change, changes the Bitcoin address beyond recognition.

So, now some Bitcoins are sent to this address before transferring them.

The end-result is that our new multiSig address now starts with 38aL and it has two transactions in the blockchain.

Sending Bitcoins from a 3 of 3 Multi-Sig Address

```
ch1408.py
import subprocess
import ast
import pybitcointools
from cfuncs1d import *
import os
def find(s, ch):
    return [i for i, ltr in enumerate(s) if ltr == ch]
def keys(baddr):
    raw = subprocess.check_output(["bitcoin-cli", "validateaddress", baddr ])
    raw = raw.replace('true', '"true"')
    raw = raw.replace('false', '"false"')
    outdict = ast.literal_eval(raw)
    pub = outdict['pubkey']
    raw = subprocess.check_output(["bitcoin-cli", 'dumpprivkey', baddr])
    pkey = raw[:-1]
```



```

    return(pub ,pkey )
baddr1 = '18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7'
baddr2 = '1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74'
baddr3 = '1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7'
(pub1 , pkey1) = keys(baddr1)
(pub2 , pkey2) = keys(baddr2)
(pub3 , pkey3) = keys(baddr3)
str0 = ["%s" , "%s" , "%s"] % (pub2 , pub1 , pub3)
raw = subprocess.check_output(["bitcoin-cli" , "createmultisig" , '3' , str0 ])
outdict = ast.literal_eval(raw)
fbaddr = outdict['address']
rScript = outdict['redeemScript']
print "Final MultiSig Bitcoin address %s" % fbaddr
#print "RedeemScript %s" % rScript
unspend = pybitcointools.history(fbaddr)
unspendl = []
i = 0
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)
spendd = unspendl[0]
#print spendd
newobject = spendd['output']
ind = newobject.index('.')
thash = newobject[0:ind]
outi = int(newobject[ind + 1:])
#print "Transaction hash %s" % thash
#print "outi %d" % outi
scr = scrpubkey(thash , outi)
#print "Script Public key %s" % scr
dbaddr = "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
bitcoin = 0.000400
str1 = [{"txid": "%s" , "vout": %d , "scriptPubKey": "%s" , "redeemScript": "%s"}] % \
    (thash , outi , scr , rScript)
str2 = {"%s": %f} % (dbaddr , bitcoin)
tbytes = subprocess.check_output(["bitcoin-cli" , "createrawtransaction" , str1 , str2])
tbytes = tbytes[:-1]
str3 = [{"txid": "%s" , "vout": %d , "scriptPubKey": "%s" , "redeemScript": "%s"}] % \
    (thash , outi , scr , rScript)
str4 = ["%s"] % pkey1
raw = subprocess.check_output(["bitcoin-cli" , "signrawtransaction" , tbytes , str3 , str4])
raw = raw.replace('true' , "true")
raw = raw.replace('false' , "false")
outdict = ast.literal_eval(raw)
fbytes = outdict['hex']
str5 = [{"txid": "%s" , "vout": %d , "scriptPubKey": "%s" , "redeemScript": "%s"}] \

```

```
% (thash , outi , scr , rScript)
str6 = ["%s"] % pkey2
raw = subprocess.check_output(["bitcoin-cli" , "signrawtransaction" , fbytes , str5 , str6 , "ALL"])
#print raw
a = find(raw , "'")
print a
fbytes1 = raw[a[2] + 1:a[3]]
str6 = [{"txid": "%s" , "vout":%d,"scriptPubKey": "%s" , "redeemScript": "%s"}] \
% (thash , outi , scr , rScript)
str7 = ["%s"] % pkey3
raw = subprocess.check_output(["bitcoin-cli" , "signrawtransaction" , fbytes1 , str6 , str7 , "ALL"])
print raw
raw = raw[:-1]
raw = raw.replace('true' , "'true'")
raw = raw.replace('false' , "'false'")
outdict = ast.literal_eval(raw)
rawtbytes = outdict['hex']
os.system("bitcoin-cli decoderawtransaction " + rawtbytes)
raw = subprocess.check_output(["bitcoin-cli" , "sendrawtransaction" , rawtbytes])
print raw
```

Output

```
Final MultiSig Bitcoin address 38aLpSoCRdDWzj8W5cbS9JzD5Av2qWB8Sj
[4, 8, 11, 696, 701, 710, 722, 729, 746, 751, 754, 819, 828, 833, 845, 855, 858,
1369, 1378, 1387, 1408, 1414, 1417, 1498]
{
  "hex":
  "010000000103a3ba98485620c300494e546f840fe19b2ffea85841b584a987f8b2e83458
b301000000fd4601004830450221009c62f3db835df910062afbc6ffcc4ce915fc6622a731
057773f53c7c36ebdd19602200bce83c9c015156206712322ae70a243b0aa6ce056a599b
a3410129c92186adc01483045022100d83c7814ba90ec2d62615dc966099744212a807c
8ea522f7478b19774451d06e02205f110f20aec3e55b385482c10dd5a506388d893d4514
384d21307e0bd503305b014730440220133ab6b64cab69bda41dc85412379384de01f3e
38fa39297400e93d24a53a27402204c3c2102998209778a568d60424c2789b7066cbbd3c
fa77d5cb6d1419306f257014c695321035693f77e6c51ebdddec9ff00f40b8c3eb8a69250
198efaf3d6522b0e80e789ee21028003e21c092ba3facaef44f92b3063389d345d03f54ac
14933f7b4707e05b36b2103fccc0fc2f33030f4bc081218671d899f883bd95c75d2a5677
2edb3d5662c59653aeffffffff01409c000000000000001976a914c847b6d84ecf8048474b19
c4a2330409ff32a1ad88ac00000000",
  "complete": true
}
{
  "txid":
  "b517db98eaf3f133526d3204d07c73d3de14e39ff9e79e99ad65212bdd732c04",
  "size": 413,
  "version": 1,
```

```

“locktime”: 0,
“vin”: [
  {
    “txid”:
      “b35834e8b2f887a984b54158a8fe2f9be10f846f544e4900c320564898baa303”,
    “vout”: 1,
    “scriptSig”: {
      “asm”: “0
        30450221009c62f3db835df910062afbc6ffc4ce915fc6622a731057773f53c7c36ebdd19
        602200bce83c9c015156206712322ae70a243b0aa6ce056a599ba3410129c92186adc[AL
        L]
        3045022100d83c7814ba90ec2d62615dc966099744212a807c8ea522f7478b19774451d0
        6e02205f110f20aec3e55b385482c10dd5a506388d893d4514384d21307e0bd503305b[ALL]
        30440220133ab6b64cab69bda41dc85412379384de01f3e38fa39297400e93d24a53a274
        02204c3c2102998209778a568d60424c2789b7066cbbd3cfa77d5cb6d1419306f257[ALL
        ]
        5321035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee210
        28003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b2103fcec
        c0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c59653ae”,
      “hex”:
        “004830450221009c62f3db835df910062afbc6ffc4ce915fc6622a731057773f53c7c36eb
        dd19602200bce83c9c015156206712322ae70a243b0aa6ce056a599ba3410129c92186a
        dc01483045022100d83c7814ba90ec2d62615dc966099744212a807c8ea522f7478b1977
        4451d06e02205f110f20aec3e55b385482c10dd5a506388d893d4514384d21307e0bd50
        3305b014730440220133ab6b64cab69bda41dc85412379384de01f3e38fa39297400e93
        d24a53a27402204c3c2102998209778a568d60424c2789b7066cbbd3cfa77d5cb6d14193
        06f257014c695321035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0
        e80e789ee21028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e0
        5b36b2103fcec0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596
        53ae”
    },
    “sequence”: 4294967295
  }
],
“vout”: [
  {
    “value”: 0.00040000,
    “n”: 0,
    “scriptPubKey”: {
      “asm”: “OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
        OP_EQUALVERIFY OP_CHECKSIG”,
      “hex”: “76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac”,
      “reqSigs”: 1,
      “type”: “pubkeyhash”,
      “addresses”: [

```

```
        "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
      ]
    }
  ]
}
b517db98eaf3f133526d3204d07c73d3de14e39ff9e79e99ad65212bdd732c04
```

Prior to understanding the code, let's verify if the above program works as advertised. For this purpose, the transaction hash value beginning with b517 shown in the last line of the output is used in the browser.

The output displayed on the screen confirms that our code works. Bitcoins from the address 38al shown on the top left have been successfully transferred to Bitcoin address 1KFz shown on the top right. The output script of this output has four opcodes only.

To repeat, Bitcoins from an address starting with 3 are transferred to an address starting with 1. At one level, both addresses are Bitcoin addresses except that a Bitcoin address beginning with 3 mandates more than one signature.

This program is a step forward as the transaction is signed with three signatures and not two.

The first change is that the method `createmultisig` is given the number 3 which mandates signing three times with the private key.

The Bitcoin address also changes and now begins with 38aL. Please use the address beginning with 3 on your screen.

The transaction hash that contains the new multi-sig address also changes. It now starts with b358 and the second Bitcoin address on the right begins with 38al. Nothing in this Bitcoin address coerces the need for three signatures or two signatures to claim ownership of the Bitcoins. Therefore, a redeem script is required.

The destination Bitcoin address will always remain the same.

The first signing with private key `pkey1` shows a value of false in the complete key. It's the same with private key `pkey2` as three signatures are required and it's been signed only twice. After the third signature with the private key `pkey3`, the complete key has a value of true. The last parameter `ALL` is optional.

For some reasons, we get an error when we parse the output returned by the second function call of `signrawtransaction`. Reading the output as a json data structure, a close relative of JavaScript also was no luck. So, in desperation, we found a `find` function on the web. This function returns the position of all the double quotes in the string or any character specified. We use these offsets to extract the hex bytes. This problem occurs only in the second signing session. This is again a kludge, we did not bother to find a more elegant solution. Thank God, code we write does not make it into the Bitcoin source.

The key hex is enclosed in two double quotes. And the data starts from the third double quote to the fourth. Variable `a[2]` denotes the starting point of the third double quote. We must read after the third quote to the start of the fourth double quote. A quick and dirty hack.

This program signs thrice. The transaction id of the third signing session is the return value of the `sendrawtransaction` method.

The output of the program gets too verbose if entirely displayed so only the third input is decoded by hand. It is shown below.

```
00
48
30450221009c62f3db835df910062afbc6ffc4ce915fc6622a731057773f53c7c36ebdd19
```

```
602200bce83c9c015156206712322ae70a243b0aa6ce056a599ba3410129c92186adc01
48
3045022100d83c7814ba90ec2d62615dc966099744212a807c8ea522f7478b19774451d0
6e02205f110f20aec3e55b385482c10dd5a506388d893d4514384d21307e0bd503305b01
47
30440220133ab6b64cab69bda41dc85412379384de01f3e38fa39297400e93d24a53a274
02204c3c2102998209778a568d60424c2789b7066cbbd3cfa77d5cb6d1419306f25701
4c
69
53
21
035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee
21
028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b
21
03fcecc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596
53
ae
```

There are three signatures instead of two. The first value in redeemScript is 0x53 which is OP_3 and not OP_2, as seen in the last example. All in all, it confirms that three private keys are required to sign and not two. The signing process remains the same, only the keys change.

CHAPTER 15

Basic Networking

The Bitcoin network is a peer to peer network where peer means all are equal. A miner may have hardware that is some zillion times more powerful than the others, but on the network, we are all equal. A peer is just a fancy word for an equal, it sounds more pompous. Bitcoin transactions are sent from one peer to another.

On the Bitcoin network, transactions are constantly moving from one Bitcoin address to another. At any point in time, every peer on the network will have a different data set as it takes some time for the network to reach a steady state. Our story today is figuring out how data is transferred amongst peers, how peers find each other and finally how a transaction is sent across, but without using the option sendrawtransaction.

Along the way, we understand the bits and bytes that transfer ownership of a Bitcoin. This chapter however, explains the Bitcoin wire protocol.

Connecting to a Bitcoin Miner or a Peer

```
ch1501.py
import socket
sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be" , 8333))
print "We have connected to our bitcoin peer"
```

Output

We have connected to our bitcoin peer.

A primer for those who know and those who don't know networking basics.

All network programming, whether on Unix or Linux or the Mac OS X and even good old Windows is known by the moniker, sockets programming. Network programming uses a protocol called TCP/IP. It all starts with the Internet Protocol (IP) where the packet size is normally 20 bytes. IP is the short form of an IP Address.

Every computer is known by an IP Address which is a number, 4 bytes long. Every phone has a number, a computer has its equivalent, an IP address. The biggest problem with the IP protocol is that it does not guarantee the delivery of packets. For this purpose, another protocol called the Transmission Control Protocol or TCP was created. TCP sits on top of IP. It is the job of TCP to reliably deliver data. Along the way, TCP also creates an abstract or virtual entity called a port number. Different types of data traffic can be layered as per the port number.

For example, all web traffic has a virtual number 80 as part of the TCP protocol. Email packets use another virtual number, 25. It is easy to distinguish different packets, web packets, email packets etc. by simply looking at the value of the port number in the TCP header.

The creators of the TCP protocol reserved all port numbers up to 1024. As the Bitcoin protocol came in very late to the party, the designers of the Bitcoin protocol had no choice but to choose a higher port number, i.e. 8333. In other words,

all Bitcoin packets will carry this number 8333 within them or in its tcp header. It like wearing a badge of honor or a way of identifying yourself.

It all started with the Unix world where they came up with a way of isolating the creation of networking packets from the programmer. They called this form of coding, sockets programming. In the bad old days of writing networking software, TCP/IP was not the only kid on the block but today it is. For some reason, sockets programming is standardized all over the solar system. It is nothing but a series of standardized C function calls. Why reinvent the wheel when the standard C libraries for sockets programming are available for every OS and phone in the world.

The module socket in the Python world is a very thin layer over the Sockets programming API. There are some zillion protocols which can be used with sockets programming, but nearly all sockets code today uses IP as the base protocol, the constant `AF_INET`. Then of all the choices available, for reliable data transfer, `SOCK_STREAM` or TCP is used. The DNS protocol uses UDP which simply sends data over without waiting to check if the other end received the data. Normally networking these days is non-error prone and packets are normally not dropped. This was not the case in the earlier days of networking in the last century.

First, an abstract entity called a socket is created where the protocol is specified using the above two numbers. The socket function returns an opaque handle to some imaginary object in the socket programming ecosystem. The socket function does nothing other than decide the protocols the rest of the code will use.

Now our program which is a peer will connect to another Bitcoin peer.

This is very different from a client server architecture where many web clients connect to a single web server. The server is the center of the universe. In a peer to peer network all are equal, unless you are a Chinese miner. We need a Bitcoin equal or a peer to connect to and the Internet has this one Bitcoin peer that many people use. The name of this Bitcoin peer is `bitcoin.sipa.be`. Another name for Dr Pieter Wuille is `sipa`.

The connect function takes only one parameter which is a tuple. A tuple can have as many parameters needed but for this function it is passed only two values.

The first value in the tuple is a web address and second variable in the tuple is the port number, 8333. When the program is executed, the connect function will send some bytes to the Bitcoin peer on port number 8333. Normally, the Bitcoin peer would send some bytes to confirm the connection and the connect call will be successful.

To be technically right, the connect function normally sends 20 bytes of IP and 20 bytes of TCP which is called the SYN packet. If the other side wants to accept the connection it will respond with a SYN-ACK packet. In the end, a final ACK is sent to acknowledge the SYN_ACK packet and the connection is complete. This process is called a three-way handshake.

The words SYN and ACK are flags in the TCP packet. We are trying to give an insight into the unpredictable world of networking.

```
ch1502.py
import socket
sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be" , 8330))
print "We have connected to our bitcoin peer"
```

Output

```
socket.error: [Errno 60] Operation timed out
```

The port number must be 8333 when establishing a connection to a miner. This program gives an error because the port number is changed to 8330.

When a SYN packet is sent at port 8330 there is no program listening/awaiting packets on the port 8330 of the server bitcoin.sipa.de. There is a program listening on port 8333. That's why, there is a time-out and an exception is thrown after a long wait.

```
ch1503.py
import socket
sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be" , 8333))
print "We have connected to our bitcoin peer"
bytesr = sock.recv(0x1000)
print "We have received %d bytes from our peer" % len(bytesr)
```

Output

```
We have connected to our bitcoin peer
We have received 0 bytes from our peer
```

The connection is successfully established, in a second, but no data comes in. There is again a long wait before the second line of display shows up.

The variable sock is used as a handle to this connection made to the Bitcoin peer. A connection must be established to send or receive data. The remote Bitcoin peer may want to send some data and the recv function is used to receive it. It takes only one parameter, the maximum number of bytes we want to receive from the other side.

A value of 0x1000 or 4096 bytes of data is specified for the second parameter. The maximum size that recv function can return is 64k of data in one go. To verify, you can read the source code in file net.cpp

The TCP protocol is not like the UDP protocol where bytes are sent in discrete entities. The TCP protocol is continuous, there is constant chatter once connected to the peer.

One central concept in sockets programming is that there is a permanent pipe created between the peers or a client – server when a connection is established. Each end sends or writes data into this pipe and the other receives or reads data from it.

If the Bitcoin peer sends 100 bytes, the recv function may return 100 bytes or only 50 bytes in the first call. The next call to the recv function may get the other 50 bytes. Everything in the network is unpredictable. The recv function has no logical understanding of any protocol, forget about the Bitcoin protocol.

A Bitcoin block size can be 1M in size today. There is a big ongoing debate on whether this limit of 1MB should be increased. If such a large block is to be read, the recv function will have to take it in blocks of 64K. This aspect must be factored while coding.

The next program worked like a charm in the first round in year 2016. That's because it followed the maxim, one logical Bitcoin packet is equal to one physical packet. Then some new Bitcoin command types were introduced and physical packets had multiple logical packets. Or, one logical packet was broken up into multiple physical packets. Our code stopped receiving these packets as it was not fine-tuned to these new updates.

You can run our code, but thou have been warned.

The Bitcoin Core versions increased from 0.12 to 0.13 to 0.14 to 0.15, the networking protocols got more complex. Our advice, try not running this code or modifying it, just sit back, put your legs on the couch and enjoying reading this chapter with a cup of coffee in the other hand. This order is for this and the next chapters only. Not our fault.

One way out is to run the bitcoind server version 0.12 on your local machine and let it download the blocks. Instead of connecting to bitcoin.sipa.be, connect to this server running at IP address 127.0.0.1. The only problem is disk space.

If this is the way forward for you, then please use version 0.11 or version 0.12. Our code on disk however shows us connecting to localhost or 127.0.0.1. We miss the simplicity of the earlier versions when it comes to networking code.

If you use bitcoin core version 0.11, the basic bitcoin.conf file must exist with the user name and password in the Bitcoin folder.

Along the way, you will get a checksum error, the code does not work as advertised, please be patient with us. There is method behind madness. We are only trying to prove a point that we should not talk to strangers halfway across the globe. In the next chapter, the sun will be shining and the birds chirping.

Sending a Version Packet and Receiving Plenty of Data from the Peer

```
ch1504.py
import socket
from cfuncs1 import *
import struct

def vers(sock):
    magic = "f9beb4d9"
    command = "version\x00\x00\x00\x00\x00"

    str1 = '62ea0000' + '0' * 164
    hash = chash(str1.decode('hex'))
    hash = hash[:4]

    plen = len(str1)/2
    str = magic + command.encode('hex') + struct.pack('<I', plen).encode('hex') + hash.encode('hex')
    str2 = str + str1
    str2 = str2.decode('hex')
    print "We are sending %d bytes" % len(str2)
    sock.sendall(str2)

sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be" , 8333))
print "We have connected to our bitcoin peer"
vers(sock)
bytesr = sock.recv(0x1000)
print bytesr
print "We have received %d Bytes which are %s" % (len(bytesr) , bytesr.encode('hex'))
```

Output

We have connected to our bitcoin peer

We are sending 110 bytes

???versionf?r7??X???? ??b/y???H/Satoshi:0.11.1/_

We have received 126 bytes which are :

```
f9beb4d976657273696f6e000000000066000000fb0fdc847211010001000000000000
0037fcac580000000001000000000000000000000000000000000000000000000000
100000000000000000000000000000000000000000000000000000000000000000
5361746f7368693a302e31312e312f175f020001
```

The Bitcoin wire or network protocol is very clear. There is a protocol to be followed for any conversation or communication. Here a simple hello will not do. The Bitcoin peer starts its conversation with a version packet.

All Bitcoin packets start with a constant 24-byte header. This header is created in the `vers` function and then sent immediately after a connection is established. Without the version packet, no Bitcoin peer will start a conversation. The salutation is mandatory.

The Bitcoin networking protocol uses little endian for integers, port numbers are however in big endian.

Let's first create the packet header. As learnt all along, every Bitcoin block starts with the magic number `f9beb4d9`, so does a network packet. Consistency thy name is Bitcoin. The protocol has many different packet types. A 12-byte packet type name or identifier follows the magic number. The packet being sent is the version packet. It starts with a string containing the words `version`. This is followed by 5 0's as the string must be 12 bytes large, null terminated. Please note that `\x00` is used instead of a 0, which is `ascii 48`.

The Bitcoin protocol calls this field `command` and so shall we, always follow orders.

Every network packet in the Bitcoin protocol has a header with four fields. The first field is the magic number and it is followed by the `command` field. Till now, 16 bytes are used up. Then, there are two integers, one for the payload length and the other for the double sha256 checksum. The total size of the header is 24 bytes. After having codified the first two fields, there are two more to go.

Following the header (24 bytes) is the packet data, which depends on the type of packet or `command` type.

In our code, the entire packet data is stored in a string variable called `str1`. The variable `str` has the 24-byte header. Together the strings, `str + str1` become the final packet to be sent out. This value `str + str1` is stored in a variable `str2`.

The actual length of the packet data stored in variable `str1` would be half of it. That's because the `str1` variable carries an encoded string and the bytes to be transmitted will be the real bytes or the decoded string. One option is to first decode the string and then taken the length. But, if we are taking the length of an encoded hex string, it will be divided by 2. Same answer at the end. The `command` variable is also an `ascii` string and it must be encoded as well.

The next requirement is a checksum of the hash value of the payload which is the entire packet minus the 24-byte header. Or in other words, just the packet data which is stored in variable `str1`. One of the advantages of storing the packet data or payload in a separate variable. It's more convenient to split the initial packet header and the payload into two separate variables, `str` and `str1`. First, the encoded string containing the payload data must be decoded. Then the double SHA hash valued is computed using the `chash` function and then only the first 4 bytes of this hash are to be consumed as a checksum.

Let's now understand the payload data. The version packet has many fields but fortunately for us, the Bitcoin peer uses only the first field called `version`. We learnt this when we sent several packets to the peer with different values in the other fields and it ignored them. There is a high probability, which is, that a later version of the Bitcoin mining software may check for more fields in the version packet. This code must be altered then. For now, the rest of payload is padded up with 0's; a total of 164 zeroes are added to maintain the size of the version packet.

The first byte checked by the peer is the version number of our Bitcoin client or peer. As per the Bitcoin wiki, this value represents a modern version of the Bitcoin wire protocol. The number of fields increases with newer version number.

Now to the last two fields of the packet header. Once again, the header starts with the magic number, followed by the `command` or version type. Then comes the length of the payload only and for it, the value of `str1` which is the length of the packet data is divided by 2. This value is packed to convert the number into a little-endian number. And finally, the sha256 byte hash value is added at the end of the string variable `str`.

The last three members of the header are encoded. Then the two strings, the header and the payload i.e. variables `str` and `str1` are concatenated and the resultant string is assigned to the variable `str2`. This string, `str2` is decoded.

The socket function `sendall` sends these decoded bytes to the remote client. The remote Bitcoin client replies with a packet, a version packet.

Our output shows both the decoded bytes and the encoded bytes and each one tells a different story. The decoded bytes say it in black and white.

To verify the outputs and maintain constant checks and balances, we install an open source program called Wireshark. This program shows the bytes being transmitted on our network.

At the opening screen, we chose the Wi-Fi interface, you chose the one you want. Then we run the above program we see lots of packets, some of them are Bitcoin packets. We click on the Protocol tab so that the Bitcoin packets line up to the top. Here, we see many packets with the protocol named Bitcoin.

Wireshark is installed with a dissector called Bitcoin, which understands and decodes the Bitcoin protocol. The raw bytes help in understanding the Bitcoin wire protocol.

The source and destination IP addresses are used to determine if the packet has been sent or received. The second pane shows various Bitcoin fields which can be inflated for more details. Selecting any field shows the raw bytes. Using Wireshark is optional but it is invaluable to us.

When we connected to the bitcoin peer `bitcoin.sipa.be`, it sent all sorts of different packets. These are more difficult to explain. We introduce a classic kludge.

At any point in time, on our hard disks we always have three version of the Bitcoin command set and the entire blockchain downloaded for emergencies like these, as well. So, we experiment with all the versions and reinstall the binaries to deduce an output.

Here, our bitcoind server for version 0.11 is started on our local machine and it has the IP address `bitcoin.sipa.be`. Our program connects to this server. Now, we get the packets in the order we want. Your outputs will vary.

A bane of networking. The output given below is the outcome of running the above program for the n th time.

Output

We have connected to our bitcoin peer

We are sending 110 bytes

???versionf??r???X???? ??,t?+_ /Satoshi:0.11.1/@????verack]???

We have received 150 Bytes which are

```
f9beb4d976657273696f6e000000000066000000bc8e1c03721101000100000000000000
c1f8ac580000000001000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
6f7368693a302e31312e312f40c3010001f9beb4d976657261636b000000000000000000
005df6e0e2
```

The same version packet is sent but the packet received is a larger one. This packet size is 150 or $126 + 26 = 150$. This packet has the header of 24 bytes as well.

The gloves are off, a packet called verack is received but with no data, just a constant header of 24 bytes. The Bitcoin server sends two separate packets, but the OS gives it as one big packet.

Output

We have connected to our bitcoin peer

We are sending 110 bytes

???versionf ?;r/??X???? ??4gR?Xv? /Satoshi:0.11.1/^????verack]???????pin????'Öko? -?

We have received 182 Bytes which are
f9beb4d976657273696f6e000000000660000008a208b3b721101000100000000000000
2ffcac58000000000100
00
6f7368693a302e31312e312f0f5e020001f9beb4d976657261636b000000000000000000
005df6e0e2f9beb4d970696e670000000000000000080000009cd2fde1ed96916b6ff82d
9f

Then a 182-byte packet is received; this third packet looks like a ping packet. The version packet is a necessary evil. In the month of October 2017, connecting to a real bitcoin miner gives a version number of Satoshi:0.15.99.

```
ch1505.py
import socket
from cfuncs1 import *
import struct

def vers(sock):
    magic = "f9beb4d9"
    command = "version\x00\x00\x00\x00\x00"

    str1 = '62ea0000' + '0' * 164
    hash = chash(str1.decode('hex'))
    hash = hash[:4]

    plen = len(str1)/2
    str = magic + command.encode('hex') + struct.pack('<I', plen).encode('hex') + hash.encode('hex')
    print "Bitcoin constant Header"
    print str
    str2 = str + str1
    str2 = str2.decode('hex')
    sock.sendall(str2)

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be", 8333))
print "We have connected to our bitcoin peer"
vers(sock)
bytesr = sock.recv(0x1000)
print bytesr
print "We have received %d Bytes which are %s" % (len(bytesr), bytesr.encode('hex'))
bytesr = sock.recv(0x10000)
print bytesr
print "We have received %s" % bytesr.encode('hex')
```

Output

```
We have connected to our bitcoin peer
Bitcoin constant Header
f9beb4d976657273696f6e00000000056000000adb7a193
???versionf?}R?r"?X???? ??[^;?50/Satoshi:0.11.1/??
We have received 126 Bytes which are
f9beb4d976657273696f6e00000000066000000977d5289721101000100000000000000
```

```
1422ad580000000000100000000000000000000000000000000000000000000000ffff000000000000001000
0000 0000000000000000000000000000000000000000ffff000000000208d945bc6883baa3530102f536174
6f7368693a302e31312e312f929b020001
????verack]??
The output shows f9beb4d976657261636b0000000000000000000005df6e0e2
```

Also, the verack command is received after running the same code some multiple times

The only change here is that one more rcv function is introduced. Earlier it was the version command, now it is a verack command. This verack packet can be ignored.

Thus, once a connection is established with the remote Bitcoin client, it keeps sending tons of packets. You will notice that the network bytes received always start with the magic number f9beb4d9.

You will have to execute this program many times to obtain the above output.

Receiving the Ping Packet

```

ch1506.py
import socket
from cfuncs1 import *
import struct

def vers(sock):
    magic = "f9beb4d9"
    command = "version\x00\x00\x00\x00\x00"
    str1 = '62ea0000' + '0' * 164
    hash = chash(str1.decode('hex'))
    hash = hash[:4]
    plen = len(str1)/2
    str = magic + command.encode('hex') + struct.pack('<l', plen).encode('hex') + hash.encode('hex')
    str2 = str + str1
    str2 = str2.decode('hex')
    sock.sendall(str2)

sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be" , 8333))
print "We have connected to our bitcoin peer"
vers(sock)
while ( True ):
    bytesr = sock.recv(0x10000)
    (magic , command, plen , checksum ) = struct.unpack("l12sll" ,bytesr[:24])
    print "Magic Number %x" % magic
    print "Command is %s" % command
    print "Payload length is %d. Length of Newpacket is %d. %r" % (plen, len(bytesr) , plen + 24 == len(bytesr))
    print "Checksum is %x\n" % checksum

```

Output

We have connected to our bitcoin peer
Magic Number d9b4bef9

```
Command is version
Payload length is 102. Length of Newpacket is 126. True
Checksum is 4f17c87c

Magic Number d9b4bef9
Command is verack
Payload length is 0. Length of Newpacket is 24. True
Checksum is e2e0f65d

Magic Number d9b4bef9
Command is ping
Payload length is 8. Length of Newpacket is 32. True
Checksum is 37fb43f4
Press Ctrl-C to quit.
The header of the packets received are displayed in an infinite for loop.
```

This program sends just one version packet. The `recv` function is placed in an infinite while loop so it is never ending. The bytes received from the `recv` function are stored in a string variable, `bytesr`. The Bitcoin header contains 24 bytes made up of four fields, each having a constant width. The `unpack` function stores the constant Bitcoin header in four variables, aptly named as `magic`, `command` `plen` and `checksum`.

Displaying the Entire Version Packet

```
ch1507.py
import socket
from cfuncs1 import *
import sys
import datetime

def vers(sock):
    magic = "f9beb4d9"
    command = "version\x00\x00\x00\x00\x00\x00"
    str1 = '62ea0000' + '0' * 164
    hash = chash(str1.decode('hex'))
    hash = hash[:4]
    plen = len(str1)/2
    str = magic + command.encode('hex') + struct.pack('<I', plen).encode('hex') + hash.encode('hex')
    str2 = str + str1
    str2 = str2.decode('hex')
    sock.sendall(str2)

def dvers(payload):
    print "Decoding Version Packet"
    (version , services ) = struct.unpack("I8s" , payload[: 12])
    print "Version returned by node is %d" % version
    print "Services %s" % services[: -1].encode('hex')
    (timestamp , addr_rcv , addr_from) = struct.unpack("Q26s26s" , payload[12:72])
    st = datetime.datetime.fromtimestamp(timestamp).strftime('%d-%B-%Y %H:%M:%S')
    print "Time Stamp is %s" % st
    (recvservices , ipaddress , ipv4address , port1 , port2) = struct.unpack("8s12s4scc" , addr_rcv)
```

```

portno = ord(port1) * 256 + ord(port2)
print "Recv Services %s" % recvservices[:::-1].encode('hex')
print "IP Address %d.%d.%d.%d" %
(ord(ipv4address[1]),ord(ipv4address[1:2]),ord(ipv4address[2:3]),ord(ipv4address[3:4]))
print "Port Number %d" % portno
(fromservices , fromipaddress , fromipv4address , fromport1 , fromport2) =
struct.unpack("8s12s4scc" , addr_from)
fromportno = ord(fromport1) * 256 + ord(fromport2)
print "Recv Services %s" % fromservices[:::-1].encode('hex')
print "IP Address %d.%d.%d.%d" %
(ord(fromipv4address[1]),ord(fromipv4address[1:2]),ord(fromipv4address[2:3]),ord
(fromipv4address[3:4]))
print "Port Number %d" % fromportno
(nonce , lengthstr ) = struct.unpack("Qc" , payload[72:81])
noncestr = "%x" % nonce
print "Nonce is %s" % noncestr
lengthstr = ord(lengthstr)
unpackstr = "%ds" % lengthstr
(useragent ) = struct.unpack(unpackstr , payload[81:81+lengthstr])
print "User Agent is %s" % useragent
newlength = 81+lengthstr
(blockheight , relay ) = struct.unpack("lc" , payload[newlength:newlength+ 4 + 1])
print "block height is %d" % blockheight
print "Relay is %d" % ord(relay)

magic = "f9beb4d9"
command = "verack\x00\x00\x00\x00\x00\x00"
commandd = command.encode('hex')
length = "%08x" % 0
checksum = chash(payload)
checksumhash = checksum[:4]
checksumhash = checksumhash.encode('hex')
str = magic + commandd + length + checksumhash
str = str.decode('hex')
sock.sendall(str)
print "VerAck Send"

def fcommand(command , length , checksum , payload):
    print
    hashp = chash(payload)
    hashp4 = hashp[:4][::-1]
    checksum = "%08x" % checksum
    if checksum != hashp4.encode('hex'):
        print "Checksum failed"
        sys.exit()
    else:
        print "Checksum Passed for %s Packet" % command
        if "version" in command:

```

```
dvers(payload)
else:
    print "Unknown Packet Type"
    sys.exit()

sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be" , 8333))
print "We have connected to our bitcoin peer"
vers(sock)
while ( True ):
    bytesr = sock.recv(0x10000)
    (magic , command, plen , checksum ) = struct.unpack("I12sII" ,bytesr[:24])
    print "Magic Number %x" % magic
    print "Command is %s" % command
    print "Payload length is %d. Length of Newpacket is %d. %r" % (plen, len(bytesr) , plen + 24 == len(bytesr))
    print "Checksum is %x" % checksum
    payload = bytesr[24:24 + plen]
    fcommand(command , plen , checksum , payload)
```

Output

```
We have connected to our bitcoin peer
Magic Number d9b4bef9
Command is version
Payload length is 102. Length of Newpacket is 126. True
Checksum is 3a6c94d4

Checksum Passed for version Packet
Decoding Version Packet
Version returned by node is 70002
Services 0000000000000001
Time Stamp is 22-February-2017 11:08:27
Recv Services 0000000000000001
IP Address 0.0.0.0
Port Number 0
Recv Services 0000000000000001
IP Address 0.0.0.0
Port Number 8333
Nonce is 9a056d87fd8aeb65
User Agent is /Satoshi:0.11.1/
block height is 184628
Relay is 1
VerAck Send
Magic Number d9b4bef9
Command is verack
Payload length is 0. Length of Newpacket is 56. False
Checksum is e2e0f65d

Checksum Passed for verack Packet
Unknown Packet Type
```


In the program, the payload is first extracted from the entire packet.

Every packet is made up of packet header plus payload data. The packet header remains the same for every packet but the payload data changes depending upon the command/packet type present in the header. The real action is in the payload data.

Now some focus on the packet length which for the version packet is 126 bytes. This packet has 24 bytes of header + 102 bytes of payload, a sum of 126. Honest confession, we had to run our code multiple times to acquire this unique set of packets.

In other words, the length of the packet - 24 = the payload length. In other words, the payload length + 24 = the length of the packet received. It's an assumption here but may not always be the case as there can be multiple logical packets in one physical packet.

Once again, variable `bytesr` is the total number of bytes received. For the version packet, it is 126. The packet header shows the payload length as 102 bytes. This difference is 24 bytes, which is the size of the constant packet header.

For the remote bitcoin peer to send a block of size 1MB, it writes 1MB of data into an imaginary pipe. It's up to the network infrastructure on the remote peer to send this data to the destined computer in chunks of data of size it deems fit. The OS network software at the other end will receive these bytes of data and the `recv` function will remit them in discrete chunks of random sizes. Also, the assumption that one packet from the `recv` function will contain one command is wrong, because the `recv` function can return 10 command packets in one go. At times, it will. There is no control over the entire process at all.

However, if certain packets get dropped along the way, the TCP protocol will make sure that the bytes get resent from the remote peer.

If the length of the bytes received is say 100 and the payload length field has a value of 400 bytes then the packet received is incomplete. We must keep reading bytes from the `recv` function until the payload size of 400. The reverse is also true. If 400 bytes are received but the size of the payload is a measly 103 bytes, it implies that multiple Bitcoin packets with different commands are received. The bytes then have to be segregated into different Bitcoin packets with different commands.

As there can be multiple commands in different packets, it is better to let a function called `fcommand` take care of the different Bitcoin commands.

The function `fcommand` takes four parameters, three of them are the fields in the header. The last parameter is the actual payload data. The payload data is 24 bytes after the start of the string and it is stored in the string variable, `payload`.

In this function `fcommand`, first the hash value of the payload is calculated, which is a series of decoded bytes. Only the first few bytes of this hash are taken. The bytes are reversed to comply with the endianness. If things do not work always use `[::-1]` to reverse the bytes.

The checksum value is given as a parameter to our function. This checksum is converted into a hex string using the modifier `%08x`. This new value is also stored in the variable called `checksum`.

Thereafter, this checksum is compared with the payload hash value we just calculated. For this task, the hash value is reversed and then stored in a variable `hashp4`. This `hashp4` string is encoded before comparing it with the checksum variable. If the checksum present in the packet header does not match the checksum of the payload of the bytes, then it is a serious problem. This should never happen as the TCP protocol guarantees that if a single bit changes the remote peer will retransmit the packet.

This error check is introduced in the program as we are paranoid of anything going wrong with networking, our fears of networking in full display. Fortunately, this code is never called.

Once the checksum matches, there are multiple if statements to understand every Bitcoin command. There are separate functions for every command and each function is called with its own payload data.

The first packet type or command name is version and the associated function `dvers` is called. The fancy in clause of python is used to check for a certain string being present.

The `dvers` function is not important as the peer on the other side ignores the data sent to it. Nevertheless, there is a chance that in the future, we may have to send all this data to the opposite peer in our version command.

The payload string is fragmented into multiple unpack functions as there are padding issues. A python problem. The first 4 bytes is an int, which is the version number. This number keeps changing all the time, but not dramatically.

The next field is an 8-byte bit field of values that determines what features are to be enabled for this connection. The Bitcoin wiki's uses such language. The protocol is silent on these values. The Wireshark dissector only uses one bit field 1 which it calls Network Node Set. Earlier we had the value as 5, now 1. So, we ignore Services. In the year 2017 at times, we received a value of d.

This is followed by an 8-byte timestamp. This explains the Q in the unpack function. The date time is displayed in English and it also reveals that this chapter was written in the year 2016. But, the code was executed again while editing the book in 2017. With new versions, displaying complicated time becomes easy but `ctime` comes handy.

Then comes two 26-byte structures that reveal the IP address and port number of both peers. It is a peer to peer connection. The unpack function breaks up each of the two 26 byte structures into variables, `addr_rcv` and `addr_from`. These strings are of type `net_addr`.

Now let's decipher these 26-byte structures one field at a time. The structure starts with the same 8 bytes of services, safely ignore. Then we have the 16 bytes for a IPv6 address which no-one uses since time started. IPv6 was supposed to replace IPv4, but did not happen. The last 4 bytes of the 16 bytes are for the IPv4 address. So the first 12 bytes are ignored and the 4 bytes are read separately. IPv6 is a revolution waiting to happen. IP addresses are 4 bytes large and are displayed as 4 numbers, from 0 to 255, with dots as separators. This is called the dotted decimal notation.

The port number is at the very end. These two bytes are stored in network byte order. In the program, they are read as individual bytes and then the value in the first byte is multiplied by 256 and second by 1. Curse of endianness.

The port number is the virtual pipe that allows all code and the sockets code to transmit the Bitcoin packets to the right destinations. The port number of the client will be a random value each time as there can be multiple connections to different peers. Multitasking is possible only because of port numbers. For local connection, the port number is 0.

The second `addr` structure is of the destination/server/remote host. The port number will always be 8333.

The `net_addr` structure should start with a timestamp which is absent in the version package. Our code does not look for the 4-byte timestamp. Then is the 8-byte nonce or a random number which keeps changing for every packet sent.

Next is a string variable called a useragent. This variable starts with a length byte which itself can vary. In our code, it is assumed that it is an integer whose value is up to 252. The only reason being, that such a small number requires one byte only to store the length. One more hack. For inputs and outputs, it may have a value of 253, so the next two bytes are a short. The length of the string can now be as large as 64K.

The origin of a User-Agent field started with browsers who identified themselves to the web server. This piece of information allowed the web servers to send customized web pages. In today's world, it applies to mobile phones especially where the browsers tell the web server to avoid an image heavy page. Plus, the screen size is also small in a mobile phone.

In the version packet, the great man Mr. Satoshi's name is seen. When we sent a version packet that has a User Agent value like `VijayMukhisaFool`, there is no reaction from the client at all. : (

The next value is the last block number or the total number of blocks received by this peer so far. Your mileage here will vary. The value in our case, is a measly 184628. This only proves that the bitcoin server is yet downloading blocks and it still has the time to converse.

Then there is a relay variable that talks about announcing relayed transactions as it will reduce network traffic. We read this value in BIP 0037. A BIP is a Bitcoin Improvement Proposals. The source code however wins over BIPs. The last BIP on GitHub as of writing this book was 145 and growing. There are rules for writing a BIP.

Now, that's not all, when a version packet is received, an acknowledgement is required in form of a verack command. The client must send a packet with the verack command. It is one of the simplest command as all that is required is a 24-byte package with no data at all. The payload length field is 0 and the checksum is calculated of the previous version packets data. The verack packet has no data and hence no checksum.

No errors here. The checksum value was seen in the fcommand function but it gets recalculated.

The chash function calculates the checksum. We take the first 4 bytes and then encode this checksum. Then, the four packet header fields are concatenated and the packet is sent over to the peer.

One cannot guess what the next packet type or command would be. The program builds on one command at a time and whenever we see a command type that is not anticipated, the program exits gracefully.

The verack packet type from the server is important now. So, if you do not get the verack packet, try running the program again. Chances are that another packet type may arrive. Also, the order of the commands received from the server are sporadic. They all are to be accounted for at some point in time and this is what we will attempt to do, down the line. We will keep adding many more if statements in the else clause.

Receiving a Verack Packet

```
ch1508.py
import socket
from cfuncs1 import *
import sys
import datetime

def vers(sock):
    magic = "f9beb4d9"
    command = "version\x00\x00\x00\x00\x00"
    str1 = '62ea0000' + '0' * 164
    hash = chash(str1.decode('hex'))
    hash = hash[:4]
    plen = len(str1)/2
    str = magic + command.encode('hex') + struct.pack('<I', plen).encode('hex') + hash.encode('hex')
    str2 = str + str1
    str2 = str2.decode('hex')
    sock.sendall(str2)

def dvers(payload):
    print "Decoding Version Packet"
    (version , services ) = struct.unpack("I8s" , payload[: 12])
    print "Version returned by node is %d" % version
    print "Services %s" % services[:: -1].encode('hex')
    (timestamp , addr_rcv , addr_from) = struct.unpack("Q26s26s" , payload[12:72])
    st = datetime.datetime.fromtimestamp(timestamp).strftime('%d-%B-%Y %H:%M:%S')
```

```
print "Time Stamp is %s" % st
(recvservices , ipaddress , ipv4address , port1 , port2) = struct.unpack("8s12s4scc" , addr_recv)
portno = ord(port1) * 256 + ord(port2)
print "Recv Services %s" % recvservices[::1].encode('hex')
print "IP Address %d.%d.%d.%d" %
(ord(ipv4address[1]),ord(ipv4address[1:2]),ord(ipv4address[2:3]),ord(ipv4address[3:4]))
print "Port Number %d" % portno
(fromservices , fromipaddress , fromipv4address , fromport1 , fromport2) = struct.unpack
("8s12s4scc" , addr_from)
fromportno = ord(fromport1) * 256 + ord(fromport2)
print "Recv Services %s" % fromservices[::1].encode('hex')
print "IP Address %d.%d.%d.%d" %
(ord(fromipv4address[1]),ord(fromipv4address[1:2]),ord(fromipv4address[2:3]),ord
(fromipv4address[3:4]))
print "Port Number %d" % fromportno
(nonce , lengthstr ) = struct.unpack("Qc" , payload[72:81])
noncestr = "%x" % nonce
print "Nonce is %s" % noncestr
lengthstr = ord(lengthstr)
unpackstr = "%ds" % lengthstr
(useragent ) = struct.unpack(unpackstr , payload[81:81+lengthstr])
print "User Agent is %s" % useragent
newlength = 81+lengthstr
(blockheight , relay ) = struct.unpack("lc" , payload[newlength:newlength+ 4 + 1])
print "block height is %d" % blockheight
print "Relay is %d" % ord(relay)

magic = "f9beb4d9"
command = "verack\x00\x00\x00\x00\x00\x00"
commandd = command.encode('hex')
length = "%08x" % 0
checksum = chash(payload)
checksumhash = checksum[:4]
checksumhash = checksumhash.encode('hex')
str = magic + commandd + length + checksumhash
str = str.decode('hex')
sock.sendall(str)
print "VerAck Send"

def dack(payload):
    print "Decoding VerAck Packet\n"

def fcommand(command , length , checksum , payload):
    print
    hashp = chash(payload)
    hashp4 = hashp[:4][::-1]
    checksum = "%08x" % checksum
    if checksum != hashp4.encode('hex'):
```

```

print "Checksum failed"
sys.exit()
else:
print "Checksum Passed for %s Packet" % command
if "version" in command:
dvers(payload)
elif "verack" in command:
dack(payload)
else:
print "Unknown Packet Type"
sys.exit()

sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be" , 8333))
print "We have connected to our bitcoin peer"
vers(sock)
while ( True ):
    bytesr = sock.recv(0x10000)
    (magic , command, plen , checksum ) = struct.unpack("I12sII" ,bytesr[:24])
    print "Magic Number %x" % magic
    print "Command is %s" % command
    print "Payload length is %d. Length of Newpacket is %d. %r" % (plen, len(bytesr) , plen + 24 == len(bytesr))
    print "Checksum is %x" % checksum
    payload = bytesr[24:24 + plen]
    fcommand(command , plen , checksum , payload)

```

Output

```

We have connected to our bitcoin peer
Magic Number d9b4bef9
Command is version
Payload length is 102. Length of Newpacket is 342. False
Checksum is 94698ac1

Checksum Passed for version Packet
Decoding Version Packet
Version returned by node is 70002
Services 0000000000000001
Time Stamp is 22-February-2017 16:26:10
Recv Services 0000000000000001
IP Address 0.0.0.0
Port Number 0
Recv Services 0000000000000001
IP Address 0.0.0.0
Port Number 8333
Nonce is 5ceb435e3fa8be23
User Agent is /Satoshi:0.11.1/
block height is 189496
Relay is 1

```

```
VerAck Send
Magic Number d9b4bef9
Command is ping
Payload length is 8. Length of Newpacket is 32. True
Checksum is 113abe2a

Checksum Passed for ping Packet
Unknown Packet Type
```

Since both entities are peers on the network, the same rules apply to both. When one sends a version packet to the remote peer, it reacts with a verack.

In the fcommand function, an elif clause is inserted to check for verack command. The associated function dack does absolutely nothing as there is no payload data to analyze but a packet is still sent over.

In the next couple of programs, we will handle only one packet type/command at a time and have an elif in the fcommand function. The whole program is given even when a small change is made only. The output however will be truncated. Keep in mind that there will be some out of turn packets types like alert.

Sending a Pong Packet Back in Response to a Ping

```
ch1509.py
import socket
from cfuncs1 import *
import sys
import datetime
def vers(sock):
    magic = "f9beb4d9"
    command = "version\x00\x00\x00\x00\x00"
    str1 = '62ea0000' + '0' * 164
    hash = chash(str1.decode('hex'))
    hash = hash[:4]
    plen = len(str1)/2
    str = magic + command.encode('hex') + struct.pack('<I', plen).encode('hex') + hash.encode('hex')
    str2 = str + str1
    str2 = str2.decode('hex')
    sock.sendall(str2)

def dvers(payload):
    print "Decoding Version Packet"
    (version , services ) = struct.unpack("I8s" , payload[: 12])
    print "Version returned by node is %d" % version
    print "Services %s" % services[:: -1].encode('hex')
    (timestamp , addr_recv , addr_from) = struct.unpack("Q26s26s" , payload[12:72])
    st = datetime.datetime.fromtimestamp(timestamp).strftime('%d-%B-%Y %H:%M:%S')
    print "Time Stamp is %s" % st
    (recvservices , ipaddress , ipv4address , port1 , port2) = struct.unpack("8s12s4scc" , addr_recv)
    portno = ord(port1) * 256 + ord(port2)
    print "Recv Services %s" % recvservices[:: -1].encode('hex')
    print "IP Address %d.%d.%d.%d" %
```

```

(ord(ipv4address[:1]),ord(ipv4address[1:2]),ord(ipv4address[2:3]),ord(ipv4address[3:4]))
print "Port Number %d" % portno
(fromservices , fromipaddress , fromipv4address , fromport1 , fromport2) = struct.unpack
("8s12s4scc" , addr_from)
fromportno = ord(fromport1) * 256 + ord(fromport2)
print "Recv Services %s" % fromservices[:1].encode('hex')
print "IP Address %d.%d.%d.%d" %
(ord(fromipv4address[:1]),ord(fromipv4address[1:2]),ord(fromipv4address[2:3]),ord
(fromipv4address[3:4]))
print "Port Number %d" % fromportno
(nonce , lengthstr ) = struct.unpack("Qc" , payload[72:81])
noncestr = "%x" % nonce
print "Nonce is %s" % noncestr
lengthstr = ord(lengthstr)
unpackstr = "%ds" % lengthstr
(useragent ) = struct.unpack(unpackstr , payload[81:81+lengthstr])
print "User Agent is %s" % useragent
newlength = 81+lengthstr
(blockheight , relay ) = struct.unpack("lc" , payload[newlength:newlength+ 4 + 1])
print "block height is %d" % blockheight
print "Relay is %d" % ord(relay)

magic = "f9beb4d9"
command = "verack\x00\x00\x00\x00\x00\x00"
commandd = command.encode('hex')
length = "%08x" % 0
checksum = chash(payload)
checksumhash = checksum[:4]
checksumhash = checksumhash.encode('hex')
str = magic + commandd + length + checksumhash
str = str.decode('hex')
sock.sendall(str)
print "VerAck Send"

def dack(payload):
    print "Decoding VerAck Packet\n"

def dping(payload):
    print "Decoding Ping Packet"
    if len(payload) == 8:
        (nonce ,) = struct.unpack("Q" , payload[:8])
        print "Nonce is %x" % nonce
        nonce = "%x" % nonce
    else:
        nonce = "0000"
        nonce = nonce.decode('hex')[:1]
        nonce = nonce.encode('hex')
        checksum = chash(nonce.encode('hex'))

```

```
checksumhash = checksum[:4]
checksumhash = checksumhash.encode('hex')
magic = "f9beb4d9"
command = "pong\x00\x00\x00\x00\x00\x00\x00\x00"
commanddd = command.encode('hex')
length = "%08x" % 8
length = length[::-1]
noncelen = len(nonce)
diff = 16 - noncelen
str = magic + commanddd + length + checksumhash + nonce + '0' * diff
print str
str = str.decode('hex')
sock.sendall(str)

def fcommand(command , length , checksum , payload):
    print
    hashp = chash(payload)
    hashp4 = hashp[:4][::-1]
    checksum = "%08x" % checksum
    if checksum != hashp4.encode('hex'):
        print "Checksum failed"
        sys.exit()
    else:
        print "Checksum Passed for %s Packet" % command
        if "version" in command:
            dvers(payload)
        elif "verack" in command:
            dack(payload)
        elif "ping" in command:
            dping(payload)
        else:
            print "Unknown Packet Type"
            sys.exit()

sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be" , 8333))
print "We have connected to our bitcoin peer"
vers(sock)
while ( True ):
    bytesr = sock.recv(0x10000)
    (magic , command, plen , checksum ) = struct.unpack("112sll" , bytesr[:24])
    print "Magic Number %x" % magic
    print "Command is %s" % command
    print "Payload length is %d. Length of Newpacket is %d. %r" % (plen, len(bytesr) , plen + 24 == len(bytesr))
    print "Checksum is %x" % checksum
    payload = bytesr[24:24 + plen]
    fcommand(command , plen , checksum , payload)
```


Checksum Passed for addr Packet
Unknown Packet Type

Every networking system has some technique to figure out who all are active on their network. The network is very unreliable and in the case of Bitcoins, anyone can enter the Bitcoin network as a peer and leave without permission. The protocol must account for this contingency. The ping command steps in here.

In TCP/IP networking, a ping packet sent by one computer gets a ping reply from the destination if it is active. The protocol is called ICMP.

Here, the remote peer sends a ping packet and the destined computer replies with a pong. If no reply packet is received, then it is assumed that the remote host does not exist and it would be removed as a valid peer. Many a times, there is no reply due to poor network connectivity.

Coming back to our code, one more else condition is introduced for the ping command type and it has a function aptly named `dping`. The ping packet simply has an 8-byte nonce which the pong packet is supposed to acknowledge and send across.

Once a ping packet is received with the nonce, the first check is performed on the payload length. It must be 8 bytes. The nonce is 8 bytes long and 16 bytes wide for an encoded hex string. The nonce generated by the remote client may not be 8 bytes large. So, trailing zeroes are added at the end to fill in.

The unpack function extracts the 8-byte nonce and the bytes are reversed for endianness. A checksum is then calculated.

Finally, the constant header of 24 bytes is created, the old nonce is added and the pong packet is sent across. The packets may not arrive in the order we want so keep sending the packets and wait and watch.

The encoding and decoding of strings need a function. Also, a function is required to reverse the bytes of an encoded string.

Handling the inv Command

```
ch1510.py
import socket
from cfuncs import *
import datetime
import sys

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def sendversionpacket(sock):
    magicbytes = "f9beb4d9"
    command = "version\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')

    str1 = '62ea0000' + '0' * 164
    checksumhash1 = chash(str1.decode('hex'))
    checksumhash1 = checksumhash1[:4]

    packetlength = len(str1)/2
    packetlengthstr = "%08x" % packetlength
    str = magicbytes + commanddecoded + reversehash(packetlengthstr) + checksumhash1.encode('hex')
    str2 = str + str1
    str2 = str2.decode('hex')
    sock.sendall(str2)

def decodeversion(payloadbytes):
    #print "Decoding Version Packet"
    (version , services ) = struct.unpack("l8s" , payloadbytes[: 12])
    #print "Version returned by node is %d" % version
    #print "Services %s" % reversehash(services.encode('hex'))
    (timestamp , addr_rcv , addr_from) = struct.unpack("Q26s26s" , payloadbytes[12:72])
    st = datetime.datetime.fromtimestamp(timestamp).strftime('%d-%B-%Y %H:%M:%S')
    #print "Time Stamp is %s" % st
```

```

(recvservices , ipaddress , ipv4address , port1 , port2) = struct.unpack("8s12s4scc" , addr_recv)
portno = ord(port1) * 256 + ord(port2)
#print "Recv Services %s" % reversehash(recvservices.encode('hex'))
#print "IP Address %d.%d.%d.%d" %
(ord(ipv4address[:1]),ord(ipv4address[1:2]),ord(ipv4address[2:3]),ord(ipv4address[3:4]))
#print "Port Number %d" % portno
(fromservices , fromipaddress , fromipv4address , fromport1 , fromport2) = struct.unpack
("8s12s4scc" , addr_from)
fromportno = ord(fromport1) * 256 + ord(fromport2)
#print "Recv Services %s" % reversehash(fromservices.encode('hex'))
#print "IPV4 address %s" % fromipv4address.encode('hex')
#print "IP Address %d.%d.%d.%d" %
(ord(fromipv4address[:1]),ord(fromipv4address[1:2]),ord(fromipv4address[2:3]),ord
(fromipv4address[3:4]))
#print "Port Number %d" % fromportno
(nonce , lengthstr ) = struct.unpack("Qc" , payloadbytes[72:81])
noncestr = "%x" % nonce
#print "Nonce is %s" % noncestr
lengthstr = ord(lengthstr)
unpackstr = "%ds" % lengthstr
(useragent ) = struct.unpack(unpackstr , payloadbytes[81:81+lengthstr])
#print "User Agent is %s" % useragent
newlength = 81+lengthstr
(blockheight , relay ) = struct.unpack("lc" , payloadbytes[newlength:newlength+ 4 + 1])
#print "block height is %d" % blockheight
#print "Relay is %d" % ord(relay)

magicbytes = "f9beb4d9"
command = "verack\x00\x00\x00\x00\x00\x00"
commanddecoded = command.encode('hex')
length = "%08x" % 0
checksum = chash(payloadbytes)
checksumhash = checksum[:4]
checksumhash = checksumhash.encode('hex')
str = magicbytes + commanddecoded + length + checksumhash
str = str.decode('hex')
sock.sendall(str)
#print "VerAck Send"

def decodeverack(payloadbytes):
    #print "Decoding VerAck Packet"
    pass

def decodeping(payloadbytes):
    #print "Decoding Ping Packet"
    if len(payloadbytes) == 8:
        (nonce ,) = struct.unpack("Q" , payloadbytes[:8])
        #print "Nonce is %x" % nonce

```

```
    nonce = "%x" % nonce
    else:
        nonce = "0000"
    nonce = reversehash(nonce)
    checksum = chash(nonce.encode('hex'))
    checksumhash = checksum[:4]
    checksumhash = checksumhash.encode('hex')
    magicbytes = "f9beb4d9"
    command = "pong\x00\x00\x00\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % 8
    lengthrev = reversehash(length)
    noncelen = len(nonce)
    diff = 16 - noncelen
    str = magicbytes + commanddecoded + lengthrev + checksumhash + nonce + '0' * diff
    str = str.decode('hex')
    sock.sendall(str)

dictionary = {0:"Error" , 1:"Transaction" , 2:"Block" , 3:"Filtered Block"}
def decodeinv(payloadbytes, length):
    print "Decoding Inv Packet"
    getdatversionbytes = payloadbytes
    (count , ) = struct.unpack("c" , payloadbytes[:1])
    count = ord(count)
    payloadbytes = payloadbytes[1:]
    if count == 253:
        (count, ) = struct.unpack("h" , payloadbytes[: 2])
        payloadbytes = payloadbytes[2:]

    print "Number Inventory Vectors is %d" % count
    for i in range ( 0 , count ):
        (type , hash , ) = struct.unpack("I32s" , payloadbytes[i * 36 : i*36 + 36])
        print "Hash Type is %s" % dictionary[type]
        print "%s" % hash.encode('hex')

    magicbytes = "f9beb4d9"
    command = "getdata\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % length
    lengthrev = reversehash(length)
    checksum = chash(getdatversionbytes)
    checksumhash = checksum[:4]
    checksumhash = checksumhash.encode('hex')
    str = magicbytes + commanddecoded + lengthrev + checksumhash + getdatversionbytes.encode('hex')
    str = str.decode('hex')
    sock.sendall(str)

def handlecommand(command , length , checksum , payload):
    checksumhash = chash(payload)
```

```

checksumhashfirstfour = checksumhash[:4]
if checksum != checksumhashfirstfour.encode('hex'):
    print "Checksum failed"
    #sys.exit()
else:
    #print "Checksum Passed for %s Packet" % command
    if "version" in command:
        decodeversion(payload)
    elif "verack" in command:
        decodeverack(payload)
    elif "ping" in command:
        decodeping(payload)
    elif "inv" in command:
        decodeinv(payload , length)
    elif "addr" in command:
        pass
    elif "alert" in command:
        pass
    else:
        print "Unknown Packet Type"
        sys.exit()

sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be" , 8333))
print "We have connected to our bitcoin peer"
sendversionpacket(sock)
while ( True ):
    bytesrecieved = sock.recv(0x10000)
    (magicnumber , command , payloadlength , checksum ) = struct.unpack("I12sII" ,bytesrecieved[:24])
    #print "Magic Number %x" % magicnumber
    print "Command is %s" % command
    #print "Payload length is %d. Length of Newpacket is %d. %r" % (payloadlength, len(bytesrecieved) ,
    payloadlength + 24 == len(bytesrecieved))
    checksumstr = "%08x" % checksum
    #print "Checksum is %s\n" % reversehash(checksumstr)
    payload = bytesrecieved[24:24 + payloadlength]
    handlecommand(command , payloadlength , reversehash(checksumstr) , payload)

```

Output

```

Command is version
Command is verack
Command is ping
Command is inv
Decoding Inv Packet
Number Inventory Vectors is 3
Hash Type is Transaction
a60b25a00f3831f0b18b41c97b2e407709e893dc7df0c70d40b0a069a43ba33a

```

```
Hash Type is Transaction
286493a23d51ea63d980acea8a39183975cc9639a3f9c87299d9c1e33bff0058
Hash Type is Transaction
533e02a9b4a5db75d454ee91e9831e0deb78c70ad89744704e52d58acb2f5180
Command is tx
Unknown Packet Type
```

This code looks flaky, but the principles remain the same. You will see a different output, though, there is a remote possibility that if you try running the program once or twice, our outputs may line up. The program results in many more transaction hashes than the ones we have shown you before it quits out.

In the program, the variable names are very large, its more to satisfy the purists who believe that a variable is not a variable unless it is 12 characters large. Also, a function reversehash is introduced which reverses the hash of an encoded string. Just a convenient way of handling a task that repeats many a times.

Now is where the fun starts. The most important task is to gain access to the transactions and blocks. The inv method is one of the most widely used commands in the Bitcoin protocol. A node uses this command to announce to its peers that it has an important entity, it can be either a transaction or block. It does not send this transaction or block, but it simply divulges this information. There is a possibility that the block available with the node maybe a 1Mb block but of no use to us. So why accept it.

The Bitcoin networking protocol can be accused of being chatty but it does not transfer large amounts of data unless it is explicitly asked for. The bitcoin protocol has defined a structure called inventory vectors that has only two members, the type of hash following and the actual hash itself.

The value of 1 in type stands for a transaction and 2 refers to a block. The other two types are normally not used. The inv command starts with a variable int that gives a count of inventory vectors following.

The c option is used to read one byte and if the value is 253, it implies that variable integer needs the value in the next 2 bytes of data. A short or the h option is used to read these two bytes and then these two bytes are removed from the payload. This is playing it safe as you will not see so many transactions in one inv packet.

The inv payload data is stored in a variable called payloadbytes after removing the length byte or bytes. Then a for loop is used to read every inv vector. The unpack function reads the type and the hash which totals up to 36 bytes in one go. One of the reasons we reduced the size of payload by 1 or 2 outside the loop was to read only the data in the loop.

The inv structure is 36 bytes large. Once again, the type field is 4 bytes and the hash field is 32 bytes long. Most of the time it is the hash value of a transaction as blocks appear once every 10 minutes, on an average. The inv structure is read each time in the loop.

There are more transactions in the Bitcoin world than blocks. Here, a dictionary is used instead of a series of if statements to display the type of the hash, it is normally transaction or block. The data is always a 32-byte hash value.

It must be noted that at the start of the function, the original payload data is stored in a variable getdataversionbytes.

When an inv command arrives, a decision is to be made. Are we interested in this transaction hash or block hash? If yes, then then an explicit request is made for the transaction and block using the hash value. This assessment is important because if the peer passed on every transaction record without any approval, the traffic would congest the network.

In response to an inv vector, a command type called getdata is created. Here it specifically asks for a transaction or a block by specifying the hash value of the transaction or block just received. The type of inv structure received is passed on to it. In our case, the payload of the inv command is blindly handed over to the getdata command as payload data. The string must be null terminated with five zeroes. The length of the payload is also given to this command.

Finally, the checksum of the variable `getdataversionbytes` is taken. The reason being that it is the data of the previous payload/ payload of the `inv` command.

Once again, the `getdata` command is used to acquire the transaction and block data. But there is a wait for an `inv` command, prior to using the command `getdata`. The `inv` command is the only way for a peer to announce his/her transaction or block hash value, thus it is the only way to find out the hashes of a peer.

If the hash value is known, then the `inv` command is not needed, the `getdata` command would suffice. How to obtain the transaction hash value is another story for another day?

Reversing a Hash

```
ch1511.py
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def ourreverse(hash):
    for i in range(0, len(hash), 2):
        print i ,
        print
    for i in range(0, len(hash), 2):
        print hash[i:i+2] ,
        print
    list1 = []
    for i in range(0, len(hash), 2):
        s = hash[i:i+2]
        list1.append(s)
    print list1
    list2 = reversed(list1)
    s = ""
    for i in list2:
        print "-", i
        s = s + i
    return s

a = reversehash("123456")
print a
b = ourreverse("123456")
print b
```

Output

```
563412
0 2 4
12 34 56
['12', '34', '56']
- 56
- 34
- 12
563412
```

The above example is a standard case showcasing the complexities of python code. Here, the code is made simple by splitting it into multiple lines. Some people still like writing python code on one line only.

The first or the inner for loop simply gives a set of even numbers to break up the string, two bytes at a time. The length of the hash value is always 32 bytes and it decides the total number of even numbers we can get. The values obtained are 0 2 and 4. The values of *i* are used to access the string, two characters at a time, eventually it allows access to the entire string or hash.

These two characters are placed into a list. The list *list1* contains three strings, 12 and 34 and 56 in our case, but in the wrong order. The *reversed* function reverses the members of the list. Finally, the reversed strings present in the list are concatenated into one string. We have the right endianness of an encoded string in this string.

The rest of networking theory and code is placed in the next chapter.

We always believe that you cannot understand Bitcoin technology unless you run code. In this chapter, you cannot run all the code the way we ran it. The next chapter will remove any checksum errors that you may receive. We have delayed explanations for a certain reason.

CHAPTER 16

More Networking

This chapter continues from where we left behind and explains the concepts of networking further. You can run the program as is, there is no need to run the bitcoin server bitcoind on your local machine. We have added lots of code to make sure that everything works with the latest version, 0.15.

The output captured from the return value of the recv function earlier is shown below.

```
Command is verack
Payload length is 0. Length of Newpacket is 56. False
```

The length of the payload is 0, so the packet size should be 24 bytes. However, the bytes received are shown as 56. It thus concludes that the recv command obviously does not understand the Bitcoin protocol at all. It sends two logical Bitcoin packets in one physical packet.

The first 24 bytes of these 56 bytes are the constant header of the verack packet. The next 32 bytes are of another packet. It could be a ping packet whose size is 32 bytes, 24 + 8 byte for the nonce, or it could be smaller part of another packet. We will never know. It will be wrong to assume that the recv function will have only one logical Bitcoin packet in one physical packet at a time.

```
ProcessMessages(verack, 0 bytes): CHECKSUM ERROR nChecksum=e2e0f65d hdr.nChecksum=8561d4da
```

The checksum sent in the previous verack and pong packets were incorrect but we remained oblivious to it for some time, as there were no errors displayed. Now, because the server and client are on the same machine, the above error message is seen. A lot can be learnt about networking when both the client and server are running on the same machine. Please check it yourself.

One of the reason for delaying the checksum is that the remote peer does not state explicitly where we went wrong in calculating it. When things do not work as advertised, we need 100% control of everything starting with the source code. Here we are also told the right answer and the wrong answer. The only way to understand code is by having everything under your control.

If we did not have access to the source code, we could never get so far.

Displaying an Unconfirmed Transaction that a Miner has Send Us

```
ch1601.py
import socket
from cfuncs import *
import datetime
import sys

def reversehash(hash):
```

```
        return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def sendversionpacket(sock):
    magicbytes = "f9beb4d9"
    command = "version\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')

    str1 = '62ea0000' + '0' * 164
    str1 = '7f110100' + '0' * 162
    checksumhash1 = chash(str1.decode('hex'))
    checksumhash1 = checksumhash1[:4]

    packetlength = len(str1)/2
    packetlengthstr = "%08x" % packetlength
    str = magicbytes + commanddecoded + reversehash(packetlengthstr) + checksumhash1.encode('hex')
    #print "Bitcoin constant Header"
    #print str
    str2 = str + str1
    str2 = str2.decode('hex')
    sock.sendall(str2)

def decodeversion(payloadbytes, checksum1):
    #print "Decoding Version Packet"
    (version, services) = struct.unpack("I8s", payloadbytes[:12])
    #print payloadbytes
    #print payloadbytes.encode('hex')
    #print "Version returned by node is %d" % version
    #print "Services %s" % reversehash(services.encode('hex'))
    (timestamp, addr_recv, addr_from) = struct.unpack("Q26s26s", payloadbytes[12:72])
    st = datetime.datetime.fromtimestamp(timestamp).strftime('%d-%B-%Y %H:%M:%S')
    #print "Time Stamp is %s" % st
    (recvservices, ipaddress, ipv4address, port1, port2) = struct.unpack("8s12s4scc", addr_recv)
    portno = ord(port1) * 256 + ord(port2)
    #print "Recv Services %s" % reversehash(recvservices.encode('hex'))
    #print "IP Address %d.%d.%d.%d" %
    (ord(ipv4address[:1]),ord(ipv4address[1:2]),ord(ipv4address[2:3]),ord(ipv4address[3:4]))
    #print "Port Number %d" % portno
    (fromservices, fromipaddress, fromipv4address, fromport1, fromport2) = struct.unpack
    ("8s12s4scc", addr_from)
    fromportno = ord(fromport1) * 256 + ord(fromport2)
    #print "Recv Services %s" % reversehash(fromservices.encode('hex'))
    #print "IPv4 address %s" % fromipv4address.encode('hex')
    #print "IP Address %d.%d.%d.%d" %
    (ord(fromipv4address[:1]),ord(fromipv4address[1:2]),ord(fromipv4address[2:3]),ord
    (fromipv4address[3:4]))
    #print "Port Number %d" % fromportno
    (nonce, lengthstr) = struct.unpack("Qc", payloadbytes[72:81])
    noncestr = "%x" % nonce
    #print "Nonce is %s" % noncestr
```

```

lengthstr = ord(lengthstr)
unpackstr = "%ds" % lengthstr
(useragent ) = struct.unpack(unpackstr , payloadbytes[81:81+lengthstr])
#print "User Agent is %s" % useragent
newlength = 81+lengthstr
(blockheight , relay ) = struct.unpack("lc" , payloadbytes[newlength:newlength+ 4 + 1])
#print "block height is %d" % blockheight
#print "Relay is %d" % ord(relay)

def decodeverack(payloadbytes , checksum1):
    magicbytes = "f9beb4d9"
    command = "verack\x00\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % 0
    checksum = chash(payloadbytes)
    checksumhash = checksum[:4]
    checksumhash = checksumhash.encode('hex')
    str = magicbytes + commanddecoded + length + checksum1
    str = str.decode('hex')
    sock.sendall(str)
    #print "VerAck Send"

def decodeping(payloadbytes , checksum1):
    #print "Decoding Ping Packet"
    checksumhash = chash(payloadbytes)
    checksumhash = checksumhash[:4]
    #print "Our Checksum is %s" % reversehash(checksumhash.encode('hex'))
    if len(payloadbytes) == 8:
        (nonce ,) = struct.unpack("Q" , payloadbytes[:8])
        #print "Nonce is %x" % nonce
        nonce = "%x" % nonce
    else:
        nonce = "0000"
        nonce = reversehash(nonce)
    magicbytes = "f9beb4d9"
    command = "pong\x00\x00\x00\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % 8
    lengthrev = reversehash(length)
    noncelen = len(nonce)
    diff = 16 - noncelen
    str = magicbytes + commanddecoded + lengthrev + checksum1 + nonce + '0' * diff
    str = str.decode('hex')
    sock.sendall(str)

dictionary = {0:"Error" , 1:"Transaction" , 2:"Block" , 3:"Filtered Block"}
def decodeinv(payloadbytes, length):
    print "Decoding Inv Packet"

```

```
getdatversionbytes = payloadbytes
(count, ) = struct.unpack("c", payloadbytes[:1])
count = ord(count)
payloadbytes = payloadbytes[1:]
if count == 253:
    (count, ) = struct.unpack("h", payloadbytes[:2])
    payloadbytes = payloadbytes[2:]

print "Number Inventory Vectors is %d" % count
for i in range (0, count):
    (type, hash, ) = struct.unpack("l32s", payloadbytes[i*36 : i*36 + 36])
    print "Hash Type is %s" % dictionary[type]
    print "%s" % hash.encode('hex')

magicbytes = "f9beb4d9"
command = "getdata\x00\x00\x00\x00\x00"
commanddecoded = command.encode('hex')
length = "%08x" % length
lengthrev = reverseahash(length)
checksum = chash(getdatversionbytes)
checksumhash = checksum[:4]
checksumhash = checksumhash.encode('hex')
str = magicbytes + commanddecoded + lengthrev + checksumhash + getdatversionbytes.encode('hex')
str = str.decode('hex')
sock.sendall(str)

def decodetx(payloadbytes):
    print "Decoding Transaction Packet"
    print "Size of Version Bytes is %d" % len(payloadbytes)
    (tversion, inputcount) = struct.unpack("lc", payloadbytes[:5])
    print "Transaction Version %d" % tversion
    inputcount = ord(inputcount)
    payloadbytes = payloadbytes[5:]
    if inputcount == 253:
        (inputcount, ) = struct.unpack("h", payloadbytes[:2])
        payloadbytes = payloadbytes[2:]
    print "Number of Inputs are %d" % inputcount
    print "Size of Version Bytes is %d" % len(payloadbytes)
    for i in range (0, inputcount):
        (txid, outputindex, scriptlen) = struct.unpack("32slc", payloadbytes[:37])
        print "Length of Version bytes %d" % len(payloadbytes)
        print "Input Number %d" % i
        print "\tTransaction ID is %s" % txid.encode('hex')
        print "\tOutput Index is %d" % outputindex
        scriptlen = ord(scriptlen)
        payloadbytes = payloadbytes[37:]
        if scriptlen == 253:
            (scriptlen, ) = struct.unpack("h", payloadbytes[:2])
```

```

payloadbytes = payloadbytes[2:]
scriptSig = payloadbytes[:scriptlen]
print "\tScriptSig Length %d" % scriptlen
print "\tScriptSig %s" % scriptSig.encode('hex')
payloadbytes = payloadbytes[scriptlen:]
(sequence) = struct.unpack("l", payloadbytes[:4])
print "\tSequence %d\n" % sequence
payloadbytes = payloadbytes[4:]
(outputcount, ) = struct.unpack("c", payloadbytes[:1])
outputcount = ord(outputcount)
payloadbytes = payloadbytes[1:]
if outputcount == 253:
    (outputcount, ) = struct.unpack("h", payloadbytes[:2])
    payloadbytes = payloadbytes[2:]
    print "No of outputs is %d" % outputcount
    for i in range (0, outputcount):
        print "--Length of Version bytes %d" % len(payloadbytes)
        print "Output Number %d" % i
        (bitcoinvalue, scriptpublickeylength) = struct.unpack("Qc", payloadbytes[:9])
        print "\tBitcoin Value %d" % bitcoinvalue
        scriptpublickeylength = ord(scriptpublickeylength)
        payloadbytes = payloadbytes[9:]
        if scriptpublickeylength == 253:
            (scriptpublickeylength, ) = struct.unpack("h", payloadbytes[:2])
            payloadbytes = payloadbytes[2:]
            print "\tScript Public Key Length %d" % scriptpublickeylength
            scriptPublicKey = payloadbytes[:scriptpublickeylength]
            print "\tScript Public Key %s\n" % scriptPublicKey.encode('hex')
            payloadbytes = payloadbytes[scriptpublickeylength:]
            (locktime) = struct.unpack("l", payloadbytes[:4])
            print "LockTime is %d" % locktime
            payloadbytes = payloadbytes[4:]
            return payloadbytes
def handlecommand(command, length, checksum, payload):
    checksumhash = chash(payload)
    checksumhashfirstfour = checksumhash[:4]
    if checksum != checksumhashfirstfour.encode('hex'):
        print "Checksum failed"
        sys.exit()
    else:
        print "Checksum Passed for %s Packet" % command
        if "version" in command:
            decodeversion(payload, checksum)
        elif "verack" in command:
            decodeverack(payload, checksum)
        elif "ping" in command:

```

```
    decodeping(payload , checksum)
    elif "inv" in command:
        decodeinv(payload , length)
    elif "tx" in command:
        decodetx(payload)
    elif "alert" in command:
        pass
    elif "addr" in command:
        pass
    elif "sendheaders" in command:
        pass
    elif "sendcmpct" in command:
        pass
    elif "feefilter" in command:
        pass
    else:
        print "Unknown Packet Type"
        sys.exit()

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be", 8333))
print "We have connected to our Bitcoin peer"
sendversionpacket(sock)
bytesrecieved = ''
cnt = 0
while (True):
    cnt = cnt + 1
    print "-----Start of While length of Packet %d. Packet Number %d" % (len(bytesrecieved), cnt)
    bytesrecieved = bytesrecieved + sock.recv(64 * 1024)
    if len(bytesrecieved) == 0:
        print "Connection reset by peer"
        sys.exit()

    (magicnumber , command, payloadlength , checksum ) = struct.unpack("l12sll" ,bytesrecieved[:24])
    print "Start of Packet Length of s is %d. Command %s" % (len(bytesrecieved) , command)

    if magicnumber != 0xd9b4bef9:
        print "Invalid Magic Number error"
        continue
    else:
        #print "Packet starts with the Magic Number "
        pass

    while len(bytesrecieved) >= payloadlength + 24:
        print "Command is %s" % command
        print "Payload length is %d. Length of Newpacket is %d. %r" % (payloadlength, len(bytesrecieved) ,
        payloadlength + 24 == len(bytesrecieved))
        checksumstr = "%08x" % checksum
```

```

print "Checksum is %s" % reversehash(checksumstr)
payload = bytesrecieved[24:24 + payloadlength]
handlecommand(command , payloadlength , reversehash(checksumstr) , payload)
bytesrecieved = bytesrecieved[24 + payloadlength:]
if len(bytesrecieved) >= 24:
    (magicnumber , command, payloadlength , checksum ) = struct.unpack("I12sII" ,bytesrecieved[:24])

```

Output

```

We have connected to our Bitcoin peer
-----Start of While length of Packet 0. Packet Number 1
Start of Packet Length of s is 24. Command version
-----Start of While length of Packet 24. Packet Number 2
Start of Packet Length of s is 150. Command version
Command is version
Payload length is 102. Length of Newpacket is 150. False
Checksum is 5aca411d
Checksum Passed for version Packet
Command is verack
Payload length is 0. Length of Newpacket is 24. True
Checksum is 5df6e0e2
Checksum Passed for verack Packet
-----Start of While length of Packet 0. Packet Number 3
Start of Packet Length of s is 24. Command sendheaders
Command is sendheaders
Payload length is 0. Length of Newpacket is 24. True
Checksum is 5df6e0e2
Checksum Passed for sendheaders Packet
-----Start of While length of Packet 0. Packet Number 4
Start of Packet Length of s is 66. Command sendcmpct
Command is sendcmpct
Payload length is 9. Length of Newpacket is 66. False
Checksum is e92f5ef8
Checksum Passed for sendcmpct Packet
Command is sendcmpct
Payload length is 9. Length of Newpacket is 33. True
Checksum is ccfe104a
Checksum Passed for sendcmpct Packet
-----Start of While length of Packet 0. Packet Number 5
Start of Packet Length of s is 119. Command ping
Command is ping
Payload length is 8. Length of Newpacket is 119. False
Checksum is f7fbc33
Checksum Passed for ping Packet
Command is addr
Payload length is 31. Length of Newpacket is 87. False
Checksum is 2e435f54
Checksum Passed for addr Packet

```

Command is feefilter
Payload length is 8. Length of Newpacket is 32. True
Checksum is e80fd19f
Checksum Passed for feefilter Packet
-----Start of While length of Packet 0. Packet Number 6
Start of Packet Length of s is 24. Command inv
-----Start of While length of Packet 24. Packet Number 7
Start of Packet Length of s is 457. Command inv
Command is inv
Payload length is 433. Length of Newpacket is 457. True
Checksum is 02b941c2
Checksum Passed for inv Packet
Decoding Inv Packet
Number Inventory Vectors is 12
Hash Type is Transaction
0ea77836725323eb4ed44179aea1d8fea4ce7ac9a82a13f12886849730f345d4
-----Start of While length of Packet 0. Packet Number 8
Start of Packet Length of s is 24. Command tx
-----Start of While length of Packet 24. Packet Number 9
Start of Packet Length of s is 1717. Command tx
Command is tx
Payload length is 226. Length of Newpacket is 1717. False
Checksum is 0ea77836
Checksum Passed for tx Packet
Decoding Transaction Packet
Size of Version Bytes is 226
Transaction Version 1
Number of Inputs are 1
Size of Version Bytes is 221
Length of Version bytes 221
Input Number 0
 Tranasaction ID is
a74e3203e5b2aa2298584b4b5fb9fd245d9764f05c1712cebe669adc8900a3cb
 Output Index is 1
 ScripSig Length 107
 ScriptSig
483045022100a6fddf1557428cbf2cd1316f8734bbbe3869865e9cf4e6bd36eefcb76169c
a0e02205bbf88dd5404321915401a1b62be3622f2f1242e731d7a5a86d314860bcac2520
12102025dad7652f0b4bcbda150377c48a3075611e8d49e7437a6a69b0c348311bccf
 Sequence 4294967295

No of outputs is 2
—Length of Version bytes 72
Output Number 0
 Bitcoin Value 17500000
 Script Public Key Length 25
 Script Public Key 76a91401836ca446072b491fc7ab040bf14fc86b51d9f088ac


```

—Length of Version bytes 38
Output Number 1
  Bitcoin Value 764525600
  Script Public Key Length 25
  Script Public Key 76a91435d7a49b636eef69e7843745248ed0968f83794688ac
LockTime is 0

```

This large program performs two important tasks. The first is to receive a logical Bitcoin packet, even though there are a multiple Bitcoin commands in one physical packet. Secondly, obtaining a transaction packet with the command tx from the remote peer.

The exit from the handlecommand function stays in the program for abundant caution as it ensures that ignored packets make a clean exit.

After connecting to the Bitcoin peer, the first function to be called is sendversionpacket. It is the same old Bitcoin peer, bitcoin.sipa.be. The server bitcoind makes for a lousy miner. The only change here is with the version number.

At the start of the code is an indefinite while loop. Before the loop, the bytesrecieved variable is initialized to a null string. This string variable is used to store not only the bytes received by the recv socket call, but also bytes received from previous socket calls, if any.

In the while loop of the earlier programs, the bytes returned from the recv call were saved into the bytesrecieved variable. This time it is handled differently. The current value of the bytesrecieved variable will be concatenated with the current bytes return by the recv function call. It is assumed that the variable bytesrecieved may have some previous data left over after one iteration of the while loop. The new bytes received at the end of the string are continuously added.

All data handled at the end of the second while loop is removed. This implies that the bytes available are the ones not taken in account. The incoming bytes in the recv function are added to these bytes before returning to the top most while loop.

In this way, we can keep track of cases where the payload bytes have multiple recv calls or when two or more logical packets come in as one physical packet.

At some point in time, when the remote Bitcoin peer closes the connection, the recv function call will return a 0. This is the time to simply exit out. The peer will give no reason for ending the communication.

In our experiments with the Bitcoin protocol, this situation normally arises once every 7000 packets, on an average. Also, if you observe the behavior of the original Bitcoin Core software, it makes 8 simultaneous connections. This routine is also visible in the output of Wireshark. Whenever a remote peer disconnects, instantly a new connection is made to another remote peer. The peer disconnects intermittently and every Bitcoin client program must have code to consider this eventuality.

We prefer to simply run the program again when the peer disconnects as the code gets more complex handling all events. We are in a learning phase now.

Back to code. As before, the unpack function is used to read the first 24-byte constant Bitcoin header. The first check is always on the magic number of packet header. If the condition fails, the next packet is read. The continue command moves the control back to the outer while loop. This generally does not happen but it's better to err on the side of caution. Nothing stops the peer from sending junk packets.

The magic number error will not occur when there is a TCP connection problem. It will only happen when someone sends fake packets. Anyone can connect to a miner and use it as a base for collecting Bitcoin blocks.

Let's assume that the number of bytes received are more than the payload + constant header size, i.e. two or more logical packets are present in one physical packet.

We enter this while loop when there are one or more logical packets in one physical packet.

Let's take a specific case. The `recv` function returns 87 bytes which are stored in the `bytesreceived` variable. If it is a ping packet of 32 bytes, where the payload is 8 bytes and its content header is 24 bytes, there are 55 extra bytes in the packet. Now, to combat this eventuality of extra bytes following, one more while loop is introduced. In the while loop, the size of the `bytesreceived` variable is reduced by 32 bytes, in this specific case.

The new packet size is now $87 - 32$ or 55 bytes. The payload size is 31. Again, the loop is executed. The process continues till the value in `bytesreceived` variable is 0.

In the while loop, the constant header is first printed. The payload which starts after the 24-byte header is extracted. The end point is the packet size. Once again, the second parameter of the slice operator is not the length of the string but a position or an offset in the string. The `handlecommand` function is the next to be called with the following parameters, the command type, the `payloadlength`, the reverse checksum and the actual payload bytes.

In this case, there are more bytes than the expected payload size. However, the `handlecommand` functions demands only the payload bytes. This is because the function handles only one logical packet at a time. The slice command thus ensures that only command's data is sliced out in the payload variable.

This is our doing now. As of now only one command is handled. The payload data plus the constant header bytes are sliced from the `bytesreceived` variable. The `bytesreceived` variable normally has a value of 0 when there are no more bytes to read. 1 physical Bitcoin packet = 1 logical Bitcoin packet.

The `bytesreceived` variable can have data for more than one command. If the `bytesreceived` variable has a length of 24 bytes, it is assumed that the 24 bytes are for the header and the payload size is 0. If the size is greater than 24, the header is first extracted to get the new payload length. In the while loop, the data is checked to have at least one valid Bitcoin command. If yes, then that command is handled. If no, then the outer while loop comes into play.

The length of the variable `bytesreceived` is reduced by the length of 1 complete Bitcoin packet. This process goes on until there are no more bytes to check.

Let's take a case scenario where the length of the `bytesreceived` variable is 1000 and the payload length is 1500 bytes. This implies that some more data is expected to arrive to fill up the packet. These bytes will be concatenated at the end.

The newer bytes from the `recv` function are added at the very end of the variable `bytesreceived` to build a complete packet.

The inner while statement is true only when there are one or more command packets in the variable `bytesreceived`. Presently, it is false.

In the function `handlecommand` one more command called `tx` is added and function `decodetx` is called. The naming convention can be ignored as we are not the experts who dynamically create objects to automatically call functions with names having command name.

All this time we have learnt how to decode a transaction, so we will quickly zip through the explanations. The print statement is our only debugger. It is advisable that if not clear, at every stage print out the length of the `payloadbytes` variable. This variable `payloadbytes` is a parameter and it contains the bytes of a single transaction.

First is the transaction version which is 4 bytes, though a char would suffice. Then comes a variable `int` for the number of inputs. This value is read as a char and then the `ord` function is used to convert it into a number. The payload variable is reduced by 5, 4 for the version, 1 for the length of inputs.

This program code will work for 253 inputs and if there are more, it moves to the next if statement. Our simple approach is to keep reducing the value in the payload variable as it makes coding much simpler.

The earlier code assumed the bytes read were from a file on disk, this time it is off a string. The main task is to reduce the bytes in the payloadbytes variable. So, no saving index positions as the read values are removed after display.

Some more explanation. A transaction may contain more than 252 inputs, so the next two bytes are read for this contingency. The real input count is read and then the payload is reduced by 2 again.

Now for reading multiple inputs, the loop comes in. The variable inputcount either way contains the final number of inputs. First the 32-byte transaction hash value bearing bitcoins is read followed by the 4-byte output index. Then the scriptlen variable is given the length of the value in scriptSig field. A single byte c is used for this purpose. On the screen, we display the running input number in variable i, the transaction id and the output index.

Based on the scriptSig length, the actual value is obtained. Again, the scriptSig and its length are displayed and then the payload variable is reduced by the value in scriptSig. The unpack function with variable length numbers is simplified using the unpack function multiple times. Towards the end, the unutilized sequence number is also read.

Presently, the code is written to read a single transaction only. But a block contains multiple transactions.

The outputs are easier than inputs. The first task is to count the number of outputs. Then in a loop, the Bitcoin value the variable output field ScriptPublicKey length and the actual ScriptPublicKey are read. After acquiring these values, the last field locktime is read. Finally, the remaining payload bytes are returned since the handlecommand function does not use it.

Each physical packet received is displayed on the screen. If the length of the packet s does not start with 0, it is counted as an incomplete packet.

The first packet of command type version is incomplete. More than 24 bytes are still required so we simply read the next series of bytes from the recv function.

The length of the payload is 102 bytes but the packet has a total length of 150 bytes. It implies that a total of 150 bytes are received including the earlier 24-byte header. The second batch received is 126 bytes. The version packet will take 126 bytes, the verack will take 24 bytes, a total of 150. The inner while loop runs twice. Our code works as an incomplete version packet is first received and then comes the data in the version packet which is tagged with the verack type in one physical packet.

The magic number error does not get called as the newly received bytes are always added at the end of the string.

The three new packets types received are sendheaders, sendcmpct and feefilter. They are handled but they do nothing.

Even though twelve transaction hashes are received, only one hash value and the data of one transaction are displayed.

Receiving a Block from a Miner

```
ch1602.py
import socket
from cfuncs import *
import datetime
import sys

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def sendversionpacket(sock):
    magicbytes = "f9beb4d9"
```

```
command = "version\x00\x00\x00\x00\x00"
commanddecoded = command.encode('hex')

str1 = '62ea0000' + '0' * 164
str1 = '7f110100' + '0' * 162
checksumhash1 = chash(str1.decode('hex'))
checksumhash1 = checksumhash1[:4]

packetlength = len(str1)/2
packetlengthstr = "%08x" % packetlength
str = magicbytes + commanddecoded + reversehash(packetlengthstr) + checksumhash1.encode('hex')
#print "Bitcoin conatnt Header"
#print str
str2 = str + str1
str2 = str2.decode('hex')
sock.sendall(str2)

def decodeversion(payloadbytes , checksum1):
    #print "Decoding Version Packet"
    (version , services ) = struct.unpack("l8s" , payloadbytes[: 12])
    #print payloadbytes
    #print payloadbytes.encode('hex')
    #print "Version returnd by node is %d" % version
    #print "Services %s" % reversehash(services.encode('hex'))
    (timestamp , addr_recv , addr_from) = struct.unpack("Q26s26s" , payloadbytes[12:72])
    st = datetime.datetime.fromtimestamp(timestamp).strftime('%d-%B-%Y %H:%M:%S')
    #print "Time Stamp is %s" % st
    (recvservices , ipaddress , ipv4address , port1 , port2) = struct.unpack("8s12s4scc" , addr_recv)
    portno = ord(port1) * 256 + ord(port2)
    #print "Recv Services %s" % reversehash(recvservices.encode('hex'))
    #print "IP Address %d.%d.%d.%d" %
    (ord(ipv4address[:1]),ord(ipv4address[1:2]),ord(ipv4address[2:3]),ord(ipv4address[3:4]))
    #print "Port Number %d" % portno
    (fromservices , fromipaddress , fromipv4address , fromport1 , fromport2) = struct.unpack
    ("8s12s4scc" , addr_from)
    fromportno = ord(fromport1) * 256 + ord(fromport2)
    #print "Recv Services %s" % reversehash(fromservices.encode('hex'))
    #print "IPV4 address %s" % fromipv4address.encode('hex')
    #print "IP Address %d.%d.%d.%d" %
    (ord(fromipv4address[:1]),ord(fromipv4address[1:2]),ord(fromipv4address[2:3]),ord
    (fromipv4address[3:4]))
    #print "Port Number %d" % fromportno
    (nonce , lengthstr ) = struct.unpack("Qc" , payloadbytes[72:81])
    noncestr = "%x" % nonce
    #print "Nonce is %s" % noncestr
    lengthstr = ord(lengthstr)
    unpackstr = "%ds" % lengthstr
    (useragent ) = struct.unpack(unpackstr , payloadbytes[81:81+lengthstr])
```

```

#print "User Agent is %s" % useragent
newlength = 81+lengthstr
(blockheight , relay ) = struct.unpack("lc" , payloadbytes[newlength:newlength+ 4 + 1])
#print "block height is %d" % blockheight
#print "Relay is %d" % ord(relay)

def decodeverack(payloadbytes , checksum1):
    magicbytes = "f9beb4d9"
    command = "verack\x00\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % 0
    checksum = chash(payloadbytes)
    checksumhash = checksum[:4]
    checksumhash = checksumhash.encode('hex')
    str = magicbytes + commanddecoded + length + checksum1
    str = str.decode('hex')
    sock.sendall(str)
    #print "VerAck Send"

def decodeping(payloadbytes , checksum1):
    #print "Decoding Ping Packet"
    checksumhash = chash(payloadbytes)
    checksumhash = checksumhash[:4]
    #print "Our Checksum is %s" % reversehash(checksumhash.encode('hex'))
    if len(payloadbytes) == 8:
        (nonce , ) = struct.unpack("Q" , payloadbytes[:8])
        #print "Nonce is %x" % nonce
        nonce = "%x" % nonce
    else:
        nonce = "0000"
        nonce = reversehash(nonce)
    magicbytes = "f9beb4d9"
    command = "pong\x00\x00\x00\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % 8
    lengthrev = reversehash(length)
    noncelen = len(nonce)
    diff = 16 - noncelen
    str = magicbytes + commanddecoded + lengthrev + checksum1 + nonce + '0' * diff
    str = str.decode('hex')
    sock.sendall(str)

dictionary = {0:"Error" , 1:"Transaction" , 2:"Block" , 3:"Filtered Block"}
def decodeinv(payloadbytes, length):
    #print "Decoding Inv Packet"
    getdatversionbytes = payloadbytes
    (count , ) = struct.unpack("c" , payloadbytes[:1])
    count = ord(count)

```

```
payloadbytes = payloadbytes[1:]
if count == 253:
    (count, ) = struct.unpack("h", payloadbytes[ : 2])
    payloadbytes = payloadbytes[2:]

    #print "Number Inventory Vectors is %d" % count
    for i in range ( 0 , count ):
        (type , hash , ) = struct.unpack("l32s" , payloadbytes[i * 36 : i*36 + 36])
        #print "Hash Type is %s" % dictionary[type]
        #print "%s" % hash.encode('hex')

    magicbytes = "f9beb4d9"
    command = "getdata\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % length
    lengthrev = reverseahash(length)
    checksum = chash(getdatversionbytes)
    checksumhash = checksum[:4]
    checksumhash = checksumhash.encode('hex')
    str = magicbytes + commanddecoded + lengthrev + checksumhash + getdatversionbytes.encode('hex')
    str = str.decode('hex')
    sock.sendall(str)

def decodetx(payloadbytes):
    #print "Decoding Transaction Packet"
    #print "Size of Version Bytes is %d" % len(payloadbytes)
    (tversion , inputcount) = struct.unpack("lc", payloadbytes[:5])
    #print "Transaction Version %d" % tversion
    inputcount = ord(inputcount)
    payloadbytes = payloadbytes[5:]
    if inputcount == 253:
        (inputcount, ) = struct.unpack("h", payloadbytes[ : 2])
        payloadbytes = payloadbytes[2:]
        #print "Number of Inputs are %d" % inputcount
        #print "Size of Version Bytes is %d" % len(payloadbytes)
        for i in range ( 0 , inputcount):
            (txid , outputindex , scriptlen ) = struct.unpack("32slc", payloadbytes[ : 37])
            #print "Length of Version bytes %d" % len(payloadbytes)
            #print "Input Number %d" % i
            #print "\tTranasaction ID is %s" % txid.encode('hex')
            #print "\tOutput Index is %d" % outputindex
            scriptlen = ord(scriptlen)
            payloadbytes = payloadbytes[37:]
            if scriptlen == 253:
                (scriptlen, ) = struct.unpack("h", payloadbytes[ : 2])
                payloadbytes = payloadbytes[2:]
                scriptSig = payloadbytes[:scriptlen]
                #print "\tScripSig Length %d" % scriptlen
```

```

#print "\tScriptSig %s" % scriptSig.encode('hex')
payloadbytes = payloadbytes[scriptlen:]
(sequence) = struct.unpack("l", payloadbytes[: 4])
#print "\tSequence %d\n" % sequence
payloadbytes = payloadbytes[4:]
(outputcount, ) = struct.unpack("c", payloadbytes[:1])
outputcount = ord(outputcount)
payloadbytes = payloadbytes[1:]
if outputcount == 253:
    (outputcount, ) = struct.unpack("h", payloadbytes[: 2])
    payloadbytes = payloadbytes[2:]
#print "No of outputs is %d" % outputcount
for i in range ( 0 , outputcount):
    #print "--Length of Version bytes %d" % len(payloadbytes)
    #print "Output Number %d" % i
    (bitcoinvalue , scriptpublickeylength) = struct.unpack("Qc", payloadbytes[:9] )
    #print "\tBitcoin Value %d" % bitcoinvalue
    scriptpublickeylength = ord(scriptpublickeylength)
    payloadbytes = payloadbytes[9:]
    if scriptpublickeylength == 253:
        (scriptpublickeylength, ) = struct.unpack("h", payloadbytes[: 2])
        payloadbytes = payloadbytes[2:]
    #print "\tScript Public Key Length %d" % scriptpublickeylength
    scriptPublicKey = payloadbytes[:scriptpublickeylength]
    #print "\tScript Public Key %s\n" % scriptPublicKey.encode('hex')
    payloadbytes = payloadbytes[scriptpublickeylength:]
    (locktime) = struct.unpack("l", payloadbytes[:4] )
    #print "LockTime is %d" % locktime
    payloadbytes = payloadbytes[4:]
    return payloadbytes

def decodegetheaders(payloadbytes):
    #print "Decoding GetHeaders Packet"
    (version , count) = struct.unpack("lc", payloadbytes[:5])
    print "Version is %d" % version
    count = ord(count)
    payloadbytes = payloadbytes[5:]
    if count == 253:
        (count, ) = struct.unpack("h", payloadbytes[: 2])
        payloadbytes = payloadbytes[2:]
    #print "Number of blocks is %d" % count
    for i in range ( 0 , count + 1 ):
        (hash , ) = struct.unpack("32s", payloadbytes[i * 32 : i*32 + 32])
        print "%s" % hash.encode('hex')

def decodeaddr(payloadbytes):
    #print "Decoding Addr Packet"
    (count , ) = struct.unpack("c", payloadbytes[:1])

```

```
count = ord(count)
payloadbytes = payloadbytes[1:]
if count == 253:
    (count, ) = struct.unpack("h", payloadbytes[2:])
    payloadbytes = payloadbytes[2:]
    for i in range ( 1 , count+1):
        (timestamp , recvservices , ipaddress , ipv4address , port1 , port2) = struct.unpack
        ("l8s12s4scc", payloadbytes[:30])
        portno = ord(port1) * 256 + ord(port2)
        st = datetime.datetime.fromtimestamp(timestamp).strftime('%d-%B-%Y %H:%M:%S')
        #print "Recv Services %s" % reversehash(recvservices.encode('hex'))
        #print "IP Address %d.%d.%d.%d" %
        (ord(ipv4address[:1]),ord(ipv4address[1:2]),ord(ipv4address[2:3]),ord(ipv4address[3:4]))
        #print "Port Number %d" % portno
        #print "Time Stamp is %s" % st
        payloadbytes = payloadbytes[30:]
        if count >= 2:
            sys.exit()
def decodeblock(payloadbytes):
    print "Decoding Block Packet"
    (bversion , prevblock , merkle , timestamp , difficulty , nonce , tcount) = \
    struct.unpack("l32s32slllc", payloadbytes[:4+32+32+4+4+4+1])
    blockhash = chash(payloadbytes[:4+32+32+4+4+4+1])
    print reversehash(blockhash).encode('hex')
    tcount = ord(tcount)
    print "Length of Payload %d" % len(payload)
    print "Block version %d" % bversion
    print "Previous Block Hash %s" % reversehash(prevblock.encode('hex'))
    print "Merkle Root %s" % reversehash(merkle.encode('hex'))
    st = datetime.datetime.fromtimestamp(timestamp).strftime('%d-%B-%Y %H:%M:%S')
    print "Block Time %s" % st
    print "Difficulty %d" % difficulty
    print "Nonce %d" % nonce
    print "Number of Transactions %d" % tcount
    payloadbytes = payloadbytes[81:]
    if tcount == 253:
        (tcount, ) = struct.unpack("h", payloadbytes[2:])
        print "Number of transactions are " , tcount
        payloadbytes = payloadbytes[2:]
        for i in range ( 1 , tcount+1):
            print "Transaction Number %d of %d" % (i , tcount)
            payloadbytes = decodetx(payloadbytes)
def handlecommand(command , length , checksum , payload):
    checksumhash = chash(payload)
    checksumhashfirstfour = checksumhash[:4]
```

```

if checksum != checksumhashfirstfour.encode('hex'):
    print "Checksum failed"
    sys.exit()
else:
    #print "Checksum Passed for %s Packet" % command
    if "version" in command:
        decodeversion(payload , checksum)
    elif "verack" in command:
        decodeverack(payload , checksum)
    elif "ping" in command:
        decodeping(payload , checksum)
    elif "inv" in command:
        decodeinv(payload , length)
    elif "tx" in command:
        decodetx(payload)
    elif "alert" in command:
        pass
    elif "addr" in command:
        pass
    elif "sendheaders" in command:
        pass
    elif "sendcmpct" in command:
        pass
    elif "feefilter" in command:
        pass
    elif "block" in command:
        print "Length %d" % (length)
        decodeblock(payload)
        exit()
    elif "getheaders" in command:
        decodegetheaders(payload)
        sys.exit()
    else:
        print "Unknown Packet Type"
        sys.exit()

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be", 8333))
#sock.connect(("5.9.7.180", 8333))
#print "We have connected to our Bitcoin peer"
sendversionpacket(sock)
bytesrecieved = ''
cnt = 0
while (True):
    cnt = cnt + 1
    #print "-----Start of While length of Packet %d. Packet Number %d" % (len(bytesrecieved), cnt)
    bytesrecieved = bytesrecieved + sock.recv(64 * 1024)

```

```
if len(bytesrecieved) == 0:
    print "Connection reset by peer"
    sys.exit()

(magicnumber , command, payloadlength , checksum ) = struct.unpack("I12sII",bytesrecieved[:24])
#print "Start of Packet Length of s is %d. Command %s" % (len(bytesrecieved) , command)

if magicnumber != 0xd9b4bef9:
    print "Invalid Magic Number error"
    continue
else:
    #print "Packet starts with the Magic Number "
    pass

while len(bytesrecieved) >= payloadlength + 24:
    #print "Command is %s" % command
    #print "Payload length is %d. Length of Newpacket is %d. %r" %
    (payloadlength, len(bytesrecieved) , payloadlength + 24 == len(bytesrecieved))
    checksumstr = "%08x" % checksum
    #print "Checksum is %s" % reversehash(checksumstr)
    payload = bytesrecieved[24:24 + payloadlength]
    handlecommand(command , payloadlength , reversehash(checksumstr) , payload)
    bytesrecieved = bytesrecieved[24 + payloadlength:]
    if len(bytesrecieved) >= 24:
        (magicnumber , command, payloadlength , checksum ) = struct.unpack("I12sII",bytesrecieved[:24])
```

Output

```
Length 998159
Decoding Block Packet
Length of Payload 998159
Block version 536870914
Previous Block Hash 0000000000000000093910d0780aa62242461d9c3db96590ae5b214b5a3dcb2
Merkle Root 19b5676b6bffc897e5f71d824e01016d3262e2c66e4c1b77fb612993db2693e6
Block Time 22-February-2017 21:56:01
Difficulty 402816659
Nonce 444687637
Number of Transactions 253
Number of transactions are 2205
Transaction Number 1 of 2205
Transaction Number 2 of 2205
```

Some of the output is truncated to avoid clutter.

When we ran this program one more time, this is one line from the output

```
Block Time 05-October-2017 16:53:47
```

The most important command is the block command which comes visiting only once every 10 minutes, on an average. The decodeblock function is called to decode the block command type. This function is much simpler as the entire block of data is given to it. The length of our block is close to the 1MB block limit.

The block version is always 1. Now the value is 536870914. When this value is converted to hex, it is 20000002, which is the signed equivalent of 1. The explorer also shows a field called version with the same value.

The Bitcoin Core simply saves these bytes to disk as the block header. The fields of the 80-byte block header are printed. We key in the previous block hash value,

```
0000000000000000093910d0780aa62242461d9c3db96590ae5b214b5a3dcb2
```

and then simply click on next block to arrive at block number 454205.

The Merkle hash value starting with 19b56 matches. The nonce is a number 444687637 and the difficulty or bits has a value of 402816659. The time is off by 5:30 hours as we are in India. Finally, the number of transactions are 2205.

Then the count/total number of transactions is read and checked to be larger than 253. The function `decodetx` is called in a for loop. The return value is saved which is the original payload. The first or topmost transaction removed. In the loop, the existing transaction is displayed and then removed and the new payload is used as the input for the next transaction to be displayed.

The same rules are followed as before, as in display one transaction, remove it from the payload and handover the remaining transactions back to the loop. The function `decodetx` displays only one transaction and return the un-displayed ones. The loop is definite as the count of transactions is available.

In this manner, Bitcoin client gets a block of data. At times, there can be a wait of 10 minutes to an hour. The Bitcoin client adds the magic number and the length of the transactions and stores this block on disk.

Output

```
Start of Packet Length of s is 55. Command addr
Command is addr
Payload length is 31. Length of Newpacket is 55. True

Decoding Addr Packet
Recv Services 0000000000000005
IP Address 94.132.217.149
Port Number 8333
Time Stamp is 23-February-2016 21:45:49
```

Randomly, an `addr` command is also received, we display this packet. It is a list of `addr` structures. An `addr` structure represents an IP address and a port number, always 8333. Later versions also have a time stamp. It is an unimportant command, as it is intermittent in nature. We ran this program for a long time before we got this output.

A node that does not advertise itself every 3 hours should be forgotten. One of the packets expected was a list of nodes that we could connect to, but it did not arrive. This program always gave only one structure in the `addr` command and the command `getheaders` for some reason also stopped being called.

How to Receive the Reject and Alert Command.

```
ch1603.py
import socket
from cfuncs import *
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def vers(sock):
    magic = "f9beb4d9"
```

```
command = "version\x00\x00\x00\x00\x00"
commandd = command.encode('hex')
str1 = '0000' + '0' * 166
hash = chash(str1.decode('hex'))
hash = hash[:4]
packetlength = len(str1)/2
packetlengthstr = "%08x" % packetlength
str = magic + commandd + reversehash(packetlengthstr) + hash.encode('hex')
str2 = str + str1
str2 = str2.decode('hex')
sock.sendall(str2)

sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be" , 8333))
print "We have connected to our bitcoin peer"
vers(sock)
bytesr = sock.recv(0x10000)
print "Length of Bytes received %d" % len(bytesr)
print bytesr
bytesr = sock.recv(0x10000)
print "Length of Bytes received %d" % len(bytesr)
print bytesr
```

Output

```
We have connected to our bitcoin peer
Length of Bytes received 24
????reject*??qJ
Length of Bytes received 42
version Version must be 31800 or greater
```

The reject command is sent when the remote peer rejects the Bitcoin command. As we mentioned earlier, the version command needs a version number larger than a certain value and here, it is just a 0. When the remote peer receives such a packet it sends back a reject command with details on which command caused the rejection. This is followed by an error message which explicitly state that the minimum version number should be 31800.

You can see that the command type is reject in the bytes displayed.

Now, the following string is sent across after connecting to the peer.

```
str1 = '62ea' + '0' * 100
```

Output

```
We have connected to our bitcoin peer
Length of Bytes received 24
????reject????
Length of Bytes received 31
versionerror parsing message
```

Only a single line in the above program is changed, which is the value of the variable `str1`. The error comes about as the command version needs to be off a certain minimum size.

The version length is replaced from 166 to a smaller number, 100 bytes. The remote peer could not parse the version command and hence the error. It just proves that the size is important, not the actual contents. So, let's make the change.

```
str1 = '62ea' + '0' * 166
```

Output

```
We have connected to our bitcoin peer
Length of Bytes received 24
???versionf??z?
Length of Bytes received 102
0?'jX?/Satoshi:0.14.0/<?
```

The version number of the peer changes. It is happy with the earlier version number.

But at the same time, an alert pops up. It is displayed below.

Output

```
We have connected to our bitcoin peer
Length of Bytes received 24
???versionf]?
Length of Bytes received 318
?3&??/Satoshi:0.14.0/=?????verack]???????alert???'????????????????/URGENT: Alert key compr?L???n?h??grade
requiredFOD e???AGk??•HA;l!,??:?:?R m???? ???????
```

This is one more nail in the miner's coffin. Alerts are very inconsistently thrown. The remote peer does some sanity checks, but the quality and diligence don't meet our expectations.

```
ch1604.py
import socket
from cfuncs1d import *
import bitcoin
import subprocess
import ast
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def find(s, ch):
    return [i for i, ltr in enumerate(s) if ltr == ch]
def sendversionpacket(sock):
    magicbytes = "f9beb4d9"
    command = "version\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    str1 = '62ea' + '0' * 166
    checksumhash1 = chash(str1.decode('hex'))
    checksumhash1 = checksumhash1[:4]
```

```
    packetlength = len(str1)/2
    packetlengthstr = "%08x" % packetlength
    str = magicbytes + commanddecoded + reversehash(packetlengthstr) + checksumhash1.encode('hex')
    str2 = str + str1
    str2 = str2.decode('hex')
    sock.sendall(str2)

def decodeversion(payloadbytes):
    pass
    magicbytes = "f9beb4d9"
    command = "verack\x00\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % 0
    checksum = chash(payloadbytes)
    checksumhash = checksum[:4]
    checksumhash = checksumhash.encode('hex')
    str = magicbytes + commanddecoded + length + checksumhash
    str = str.decode('hex')
    sock.sendall(str)

def computerawtransaction():
    mybitcoinaddress = '1Lg57gLSp8HYg6W41kwddNXlik6oUZ3p3n'
    destinationbitcoinaddress = "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
    bitcoinvalue = 0.000154
    pkey = "L2T9wdbjAvyiXQnm27WPNR323Ce8cDkDkMARQx125CQZ43gHG4yR"
    listspendandunspend = bitcoin.history(mybitcoinaddress)
    unspendlist = []
    i = 0
    for obj in listspendandunspend:
        if 'spend' not in obj.keys():
            unspendlist.append(obj)
    print "Number of unspent outputs are %d" % len(unspendlist)
    spenddictionary = unspendlist[0]
    newobject = spenddictionary['output']
    indexofcolon = newobject.index(':')
    transactionhash = newobject[0:indexofcolon]
    outputindex = int(newobject[indexofcolon + 1:])
    scriptpublickey = scrpubkey(transactionhash, outputindex)
    str0 = [{"txid": "%s", "vout": %d}] % (transactionhash, outputindex)
    str1 = '{ "s" : %f} ' % (destinationbitcoinaddress, bitcoinvalue)
    raw = subprocess.check_output(["bitcoin-cli", "createrawtransaction", str0, str1])
    firstoutput = raw[:-1]
    str3 = [{"txid": "%s", "vout": %d, "scriptPubKey": "%s", "redeemScript": ""}] % (transactionhash,
    outputindex, scriptpublickey)
    str4 = ["%s"] % pkey
    str5 = "ALL"
    raw1 = subprocess.check_output(["bitcoin-cli", "signrawtransaction", firstoutput, str3, str4, str5])
```

```

raw1 = raw1.replace('true', '"true"')
outdict = ast.literal_eval(raw1)
rawoutput = outdict['hex']
raw2 = subprocess.check_output(["bitcoin-cli", "decoderawtransaction", rawoutput])
print raw2
a = find(raw2, '"')
txid = raw2[a[2] + 1:a[3]]
print "Reverse Transaction ID calculated by us %s" % reversehash(txid)
#print rawoutput
return ( reversehash(txid) , rawoutput)

def sendainvcommand():
    print "-----Sending a inv package"
    magicbytes = "f9beb4d9"
    command = "inv\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % 37
    lengthrev = reversehash(length)
    count = '01'
    invtype = '01000000'
    reversethash = reversehash(thash)
    print "Reverse Hash is %s" % reversethash
    str1 = count + invtype + reversethash
    checksum = chash(str1.decode('hex'))
    checksumhash = checksum[:4]
    checksumhash = checksumhash.encode('hex')
    #print checksumhash
    str = magicbytes + commanddecoded + lengthrev + checksumhash
    str2 = str + str1
    sock.sendall(str2.decode('hex'))

def decoding(payloadbytes):
    if len(payloadbytes) == 8:
        (nonce,) = struct.unpack("Q", payloadbytes[:8])
        nonce = "%x" % nonce
    else:
        nonce = "0000"
        nonce = reversehash(nonce)
        checksum = chash(nonce.encode('hex'))
        checksumhash = checksum[:4]
        checksumhash = checksumhash.encode('hex')
        magicbytes = "f9beb4d9"
        command = "pong\x00\x00\x00\x00\x00\x00\x00\x00"
        commanddecoded = command.encode('hex')
        length = "%08x" % 8
        lengthrev = reversehash(length)
        noncelen = len(nonce)
        diff = 16 - noncelen

```

```
    str = magicbytes + commanddecoded + lengthrev + checksumhash + nonce + '0' * diff
    str = str.decode('hex')
    sock.sendall(str)
    sendainvcommand()

dictionary = {0:"Error" , 1:"Transaction" , 2:"Block" , 3:"Filtered Block"}

def decodegetdata(payloadbytes):
    (count , ) = struct.unpack("c" , payloadbytes[:1])
    count = ord(count)
    payloadbytes = payloadbytes[1:]
    if count == 253:
        (count, ) = struct.unpack("h" , payloadbytes[: 2])
        payloadbytes = payloadbytes[2:]

    print "Number Inventory Vectors is %d" % count
    for i in range ( 0 , count ):
        (type , hash , ) = struct.unpack("I32s" , payloadbytes[i * 36 : i*36 + 36])
        print "Hash Type is %s" % dictionary[type]
        print "Transaction hash in getdata %s" % hash.encode('hex')

    print "-----Sending a tx package"
    magicbytes = "f9beb4d9"
    command = "tx\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    tbytes = rawoutput
    #print tbytes
    tbytes = tbytes.decode('hex')
    length = "%08x" % len(tbytes)
    #print "Length of raw bytes is " , length
    lengthrev = reverseahash(length)
    checksum = chash(tbytes)
    checksumhash = checksum[:4]
    checksumhash = checksumhash.encode('hex')
    #print "Checksum of message is %s" % checksumhash
    str = magicbytes + commanddecoded + lengthrev + checksumhash
    str = str.decode('hex')
    str2 = str + tbytes
    sock.sendall(str2)

def handlecommand(command , length , checksum , payload , cnt):
    if "version" in command:
        decodeversion(payload)
    elif "ping" in command:
        decodeping(payload)
    elif "getdata" in command:
        decodegetdata(payload)

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#sock.connect(("bitcoin.sipa.be", 8333))
```



```

sock.connect(("5.9.7.180", 8333))
sendversionpacket(sock)
bytesrecieved = ''
cnt = 0
(thash , rawoutput) = computerastransaction()
print "Transaction hash is %s" % thash
#print "Raw Output %s" % rawoutput
while (True):
    bytesrecieved = bytesrecieved + sock.recv(64 * 1024)
    (magicnumber , command, payloadlength , checksum) = struct.unpack("I12sII", bytesrecieved[:24])
    while len(bytesrecieved) >= payloadlength + 24:
        checksumstr = "%08x" % checksum
        payload = bytesrecieved[24:24 + payloadlength]
        cnt = cnt + 1
        handlecommand(command , payloadlength , reversehash(checksumstr) , payload , cnt)
        bytesrecieved = bytesrecieved[24 + payloadlength:]
        if len(bytesrecieved) >= 24:
            (magicnumber , command, payloadlength , checksum) = struct.unpack("I12sII", bytesrecieved[:24])

```

Output

Number of unspent outputs are 3

```

{
    "txid":
    "cdf3dcc61402dc75be9a8415ba265338f84ec1bf402b551ecfb000a5675a1599",
    "hash":
    "cdf3dcc61402dc75be9a8415ba265338f84ec1bf402b551ecfb000a5675a1599",
    "size": 192,
    "vsize": 192,
    "version": 1,
    "locktime": 0,
    "vin": [
        {
            "txid":
            "6ababf0d54df9d25e9acf8b1eae2256c8151fb22d48950ff1ea4af4e06d30e1c",
            "vout": 0,
            "scriptSig": {
                "asm":
                "3045022100f209c9abc210a44e69f531f1b56115f3dd20f07f3855c959d71185667135d4
                4302201c844f2c9d464a2e2d51640864df3be52bab49e29d8871bb2d5a1b99db0fbf1f[A
                LL]038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3",
                "hex":
                "483045022100f209c9abc210a44e69f531f1b56115f3dd20f07f3855c959d71185667135
                d44302201c844f2c9d464a2e2d51640864df3be52bab49e29d8871bb2d5a1b99db0fbf1f
                0121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3"
            },

```

```
        "sequence": 4294967295
      }
    ],
    "vout": [
      {
        "value": 0.00015400,
        "n": 0,
        "scriptPubKey": {
          "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
OP_EQUALVERIFY OP_CHECKSIG",
          "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
          "reqSigs": 1,
          "type": "pubkeyhash",
          "addresses": [
            "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
          ]
        }
      }
    ]
  }
}
```

Output

Reverse Transaction ID calculated by us 99155a67a500b0cf1e552b40bfc14ef8385326ba15849abe75dc0214c6dcf3cd
Tranasaction hash is 99155a67a500b0cf1e552b40bfc14ef8385326ba15849abe75dc0214c6dcf3cd

-----Sending a inv package

Reverse Hash is cdf3dcc61402dc75be9a8415ba265338f84ec1bf402b551ecfb000a5675a1599

Number Inventory Vectors is 1

Hash Type is Transaction

Transaction hash in getdata cdf3dcc61402dc75be9a8415ba265338f84ec1bf402b551ecfb000a5675a1599

-----Sending a tx package

The program sends raw transaction bytes to a miner without using the sendrawtransatcion command. Two programs written earlier are merged to one large program. Also, a connection is established to a hard-coded IP address.

First, the computeraawtransaction function is called before entering the main while loop. This function uses code for sending a transaction with bitcoin-cli, as seen in the previous chapter. As before, the Bitcoin addresses starting with 1Lg57 is the source of the Bitcoins and the destination Bitcoin address starts with 1KF. The private key also remains the same, starting with L2T9. Prior to this, the history command is used to obtain all unspent transactions.

The command createrawtransaction creates a transaction and then the command signrawtransaction is used to sign it. The method decoderawtransaction displays this transaction.

One method to send the transaction to one Bitcoin peer/miner is to use the sendrawtransaction who would then send it another miner. This process would keep repeating. Here, in our program, we send the transaction (without the above function) using our newly minted knowledge of the Bitcoin wire protocol.

The raw transaction bytes are decoded and then displayed and stored in variable raw2. The txid field is extracted using the double inverted comma trick, it is the transaction hash value of our newly created transaction but not mined.

In our case, it begins with cdf3d, your mileage will vary. Just a gentle reminder here, first shut down the Bitcoin-Qt and start the bitcoind as a daemon.

Also, the blockchain explorer is kept handy. Key in this transaction hash starting with cdf3d in the explorer and check if they have the same Bitcoin addresses.

The hex field contains the signed raw transaction bytes. These returned transaction bytes are stored in the variable `rawoutput`. The transaction hash stored in variable `txid` is also returned. Thereafter, these transaction hash bytes are reversed using our function `reversehash` and the values are returned as a tuple. All this code has been explained earlier.

The raw transaction bytes and the transaction hash values that are to be sent across are set but prior to that comes the mandatory version packet. In the function `handlecommand`, only three command types are monitored. These are version, ping for verack and pong commands. These are the crucial housekeeping commands.

Earlier, the remote peer responded to these packets with an `inv` command. So, as a peer the same is expected from our end as well. The remote peer is informed that there is a new transaction hash from our end, it is immune to how the hash is created and who has sent it. An `inv` packet is sent in our case, after the `pong` packet. It is to inform the peer that a transaction with a certain hash value can be sent. Then is a wait game. The peer may/may not reply depending on the relevance of the packet to it.

In the `sendainvcommand` function, first the `inv` packet is created. It starts with the 24-byte constant header keeping the length of the command string to 12 bytes, and a string `inv` followed by 9 0's. The size of the payload is hardcoded to 37 bytes as there is only one inventory vector to be sent across. The length is reversed as well. Makes coding simpler.

The first byte of the `inv` packet is a variable `int` for the number of inventory vectors following. A single byte suffices to store the value of 1. Each inventory vector starts with the type of hash, 1 for transaction and then the actual hash value. The transaction hash is needed here as the remote host is asked if this transaction hash value can serve any purpose for it. It is the most crucial piece of information and it must be sent across. The checksum of these 37 bytes is taken without the constant packet header. As before, variable `str` stores the constant header, variable `str1` has the `inv` payload data.

Subsequently, the initial packet header and the `inv` packet are concatenated and then sent to the remote Bitcoin host.

The peer on receiving the `inv` command checks whether this transaction hash is worthwhile, and accordingly, it sends a `getdata` command packet. When the `getdata` command arrives, the function `decodegetdata` gets called where all the inventory vectors are printed. In our specific case, we receive only 1 as we sent only one `inv` vector. The same transaction hash starting with cdf3 is sent back, but in reverse order.

Now that the peer is interested, it's time to send a `tx` command packet. The command string is encoded as it had been decoded earlier. Please count the number of 0's in the variable `command`. These transaction bytes calculated by `bitcoin-cli` method are stored in the variable `rawoutput` and they are decoded.

Now is the time to find the length of the transaction bytes, which is the payload length. And for the last time, these bytes are reversed after converting them to a string. The checksum is the checksum of the transaction bytes. And, it's time to decode the initial packet header present in variable `str`. The transaction bytes stored in the variable `tbytes` is already decoded.

After sending the transaction, you can wait for a reaction from the Bitcoin peer, but there will be none. Nevertheless, check the website blockchain.info, and our transaction id will be seen. This proves that a Bitcoin transaction is sent across, thus validating our claim of successfully mimicking the works of the option `sendrawtransaction`.

Once one miner has access to our transaction hash, it spreads it to other miners. Remember, it takes some time or latency for the entire lot of miners to receive our transaction. And in some time, our transaction hash will reach every miner in vast Bitcoin network.

The `sendheaders` command type is a new addition. The node announces new blocks by using the `headers` command and not the `inv` command. The `sendcmpct` command talks of announcing a new block using the new `cmpctblock`

message. Since this command is ignored, the miner does not send a `cmpctblock` command type. The `feefilter` command encodes a minimal fee in Satoshi per 1000 bytes. The alert message has a pretty complex field format for a simple alert to be displayed in the field called `StatusBar`. The other Bitcoin packet types were never sent and hence not mentioned.

Back to basics.

Why have we used programs that have a checksum `mis_match`? We did it on purpose. We were merrily sending packets to a bitcoin server on mars but got no response. We thought the network was done, the miner went rogue or the sun rose from the west. We send packets, silence is the only response we get. How do we figure out where we went wrong.

We then realized that our understanding of Bitcoin was very shallow. The same bitcoin server we use, is the very same Bitcoin server that all the miners use. We then ran `bitcoind`, the bitcoin miner on our machine and made our networking client programs connect to `localhost` instead. Then we realized that we were calculating the checksum the wrong way.

This is why we chose Bitcoin Blockchain technology over AI even though Artificial Intelligence is bigger than Blockchain technology. The problem with AI is that it will always be a black box, beyond a certain level you cannot understand how the AI works its magic. As Bitcoin and the other crypto-currencies are open source, we have no secrets between us. We have been bought up not on C, not on assembler but pure machine language. This is the world we love.

CHAPTER 17

Hashes SHA0 and SHA1

Writing this chapter and the next one was extremely difficult as even Google failed in giving us answers to some of our very basic questions. We assume you know nothing about cryptography and you are as naïve as we were at the beginning of this chapter.

With this chapter, we start our ambitious journey into the magical world of cryptography. This chapter and the next one explains the two hashes used heavily in Bitcoins. The first and foremost is the SHA-256 hash and the next one is hash RIPEMD-160. Then, the following two chapters are on Elliptic Curve Cryptography (ECC).

In the year 1990, one of the founding fathers of cryptography, Ronald Rivest invented the first hashing algorithm called MD4. Mr. Rivest is the R of RSA, the company that invented public key cryptography. Bitcoins use a variant of this concept called Elliptic Curve Cryptography, which is explained in the forthcoming chapters.

MD4 had lots of security problems, so in 1992, Mr. Rivest came out with an improved version called MD5. In 1993, the National Security Agency, NSA (that spies on the whole world including fellow Americans) came out with their own hashing algorithm called SHA or Secure Hash Algorithm.

SHA was very similar in design to MD4 and MD5. You should know that most hashing algorithms borrow a lot from the MD5 family of hashes. In 1995, the NSA detected one big security flaw in SHA, which made them change the name, SHA to SHA0. The newer algorithm with one modification was called SHA-1. People assumed that SHA had some trapdoor since it was invented by the NSA but nothing was found. Nevertheless, SHA will always be looked on with suspicion.

In August 2002, SHA-1 was replaced with SHA-2 or SHA-256. The Bitcoin world uses the SHA256 hash algorithm extensively. The framework of the trust machine is built on the strong foundations of the SHA hash concepts. The document that contains the design of the SHA family is called FIPS or Federal Information Processing Standards 180 - 1. The latest version is FIPS 180 - 4.

In this chapter, we start with SHA0, then understand the loopholes in SHA0 and then move on to SHA1. From SHA1 to SHA256 is a bigger step. RIPEMD-160 is very different from SHA256. We wonder why Mr. Satoshi chose one US standard and one European standard in his design.

Logical start to this chapter is with MD4 and MD5 and we explain it with actual code.

MD4

```
ch1701.py
import hashlib
s1 =
'839c7a4d7a92cb5678a5d5b9eea5a7573c8a74deb366c3dc20a083b69f5d2a3bb3719dc
69891e9f95e809fd7e8b23ba6318edd45e51fe39708bf9427e9c3e8b9'.decode('hex')
h1 = hashlib.new('md4', s1).hexdigest()
```

```
print h1
s2 =
'839c7a4d7a92cbd678a5d529eea5a7573c8a74deb366c3dc20a083b69f5d2a3bb3719dc
69891e9f95e809fd7e8b23ba6318edc45e51fe39708bf9427e9c3e8b9'.decode('hex')
h2 = hashlib.new('md4', s2).hexdigest()
print h2
print h1 == h2
print s1 == s2
```

Output

```
4d7e6a1defa93d2dde05b45d864c429b
4d7e6a1defa93d2dde05b45d864c429b
True
False
```

The MD4 algorithm is highly unreliable as two different sets of characters can generate the same hash value.

The good old hashlib python library is imported to convert a block of bytes into a md4 hash.

The md4 hash is not available as a built-in function because it is not used anymore. So, the new function from module hashlib is called with the name of the hashing algorithm. This function is also given the s1 variable, which has the raw decoded bytes to be hashed. This hash value is then displayed. The original value serves no purpose.

Then another block of data is given to the same function which looks the same but has some subtle differences. In variable s1, the bytes edd4 is replaced by edc4. The last output confirms that the strings, s1 and s2 are different.

The hexdigest function gives the hex representation of the calculated hash value.

On executing the code, the MD4 hash is produced and it is the same for both the values. This is a sign of disaster as two different blocks of data cannot generate the same hash. So, if the same algorithm is applied in Bitcoins, then the bytes of two different 80-byte Bitcoin block headers would also produce the same hash. The entire foundations on which the concept of the blockchain rests would collapse in the blink of an eye.

Let's move on to MD5.

MD5

```
ch1702.py
import hashlib
s1 =
'd131dd02c5e6eec4693d9a0698aff95c2fcab58712467eab4004583eb8fb7f8955ad3406
09f4b30283e4888325f1415a085125e8f7cdc99fd91dbdf280373c5bd8823e3156348f5b
ae6dacd436c919c6dd53e2b487da03fd02396306d248cda0e99f33420f577ee8ce54b670
80a80d1ec69821bcb6a8839396f9652b6ff72a70'.decode('hex')
h = hashlib.md5(s1).hexdigest()
print(h)
s2 =
'd131dd02c5e6eec4693d9a0698aff95c2fcab50712467eab4004583eb8fb7f8955ad3406
09f4b30283e4888325f1415a085125e8f7cdc99fd91dbd7280373c5bd8823e3156348f5b
ae6dacd436c919c6dd53e23487da03fd02396306d248cda0e99f33420f577ee8ce54b670
80280d1ec69821bcb6a8839396f965ab6ff72a70'.decode('hex')
```

```
h1 = hashlib.md5(s2).hexdigest()
print(h1)
print h == h1
print s1 == s2
```

Output

```
79054025255fb1a26e4bc422aef54eb4
79054025255fb1a26e4bc422aef54eb4
True
False
```

The MD5 hashing algorithm is more popular than MD4; there are in-built functions for it in the hashlib library so, the new function is not used. First, a function called md5 in the hashlib module is given a block of decoded data to compute the hash value. The hexdigest function is then used to return a hex representation of this hash value. The md5 function is called twice with two different values in variable s1 and s2. Though the strings are different, they give the same result. Even md5 is not reliable anymore.

So back to the drawing board. Let's understand the source code of SHA0. Explaining everything about SHA0 hash in one large program will leave you with a splitting headache. Therefore, the computation of the SHA0 hash is broken into multiple small programs first and then united into one large program.

Numbers in Little Endian Look Very Different then the Same Number in Big Endian

```
ch1703.py
import struct
v = (0x67452301 , 0xEFCDA89 , 0x98BADCFE , 0x10325476 , 0xC3D2E1F0)
for i in v:
    v1 = struct.pack("<I" , i)
    print v1.encode('hex')
```

Output

```
01234567
89abcdef
fedcba98
76543210
f0e1d2c3
```

In every hash programs, some variables are given an initial value. The initial values are never 0's but some randomly chosen numbers. These values are called Initialization Vectors or IV's, another word for a very random number. The designers of the hash algorithm may use these so called random values as a backdoor or trapdoor to have control over the generated hashes.

In SHA0 algorithm, the initial values look like 0x67452301. But, in the little-endian format (reversing the bytes), the value is 01234567. There is nothing sinister about these values at all when viewed in the little-endian format. The first two display the hex digits forwards, the next 2 display the same digits backwards and the last is a combination of 0123 and so on.

In the program, a tuple v is created with some IV's that the sha0 hash uses. The python code simply uses the pack function where < stands for little-endian. The variable i represents each value stored in the tuple. This approach is the fastest way to convert a big-endian number into a little-endian string.

The unpack function converts the string to a number, it's been used many times before. As little-endian numbers, they look innocuous.

Taking Simple Numbers and Making them Look Imposing

```
ch1704.py
import math
v = (2 , 3 , 5 , 10)
for i in v:
    s = math.sqrt(i)
    s1 = pow(2 , 30 ) * s
    print s1
    print "%08x:%d" % (s1,s1)

    print pow(2 , 30 )
```

Output

```
1518500249.99
5a827999:1518500249
1859775393.38
6ed9eba1:1859775393
2400959708.75
8f1bbcdc:2400959708
3395469782.82
ca62c1d6:3395469782
1073741824
```

In the SHA0, four constants are used in a for loop. These constants again are random numbers and they have no special meaning. There must be no hidden design on why these numbers are chosen. These constants are seen in the output of the above program.

What better numbers than the square root of 2 and 3 and 5 and 10. Plus, these square roots are multiplied by 2 raised to 30. No idea why the square root of 10 was chosen and not 7 as it would then make them, prime numbers. Why raise the floating-point number to the power of 30 and not 29 or 31?

In the code, tuple v is initialized to these four values, namely 2,3,5 and 10. First the square root of each of the above numbers is calculated. The sqrt function returns a float and the result is stored in the variable s. The pow function which calculates the power of the first parameter 2 to the second parameter 30 as in 2^{30} gives a constant value of 1073741824. The hex value of this number is a little less imposing, 0x40000000.

The s1 variable is a float and converted to a hex value. This is achieved by simply ignoring the numbers after the decimal point.

The s1 variable is a float whose value is 1518500249.988025 for the square root of 2 multiplied by 2 raised to 30. The decimal portion is removed so the value is 1518500249 as a decimal number or 5a827999 as a hex number. This explains the first output value.

Revision of Bitwise Anding, Oring and Xoring

```
ch1705.py
i = 5
j = 6
print "i %s" % format(i, '04b')
print "j %s" % format(j, '04b')
print "AND i & j %s" % format((i & j), '04b')
print "OR i | j %s" % format((i | j), '04b')
print "XOR i ^ j %s" % format((i ^ j), '04b')
```

Output

```
i 0101
j 0110
AND i & j 0100
OR i | j 0111
XOR i ^ j 0011
```

Let's take a 2-minute detour to simply revise some basics of Bitwise operations, knowing that most of you are well conversant with it by now.

The variables *i* and *j* are initialized to values 5 and 6 and the format function display the values in these variables in a certain manner. The second parameter of the format function decides the way the first parameter must be displayed. So, the output will be in binary because in the second parameter, *b* is given as the special format character. Once again, the *b* at the end means binary, the 0 at the beginning refers to display of leading zeroes and the 4 represents the number of binary digits to be displayed.

The bits of *i* and *j* line up as a truth table. This table reveals the three possibilities the bit combinations can have. These are a 0 and 0, 1 and 1, 1 and 0 and 0 and 1.

The *&* in Python stands for the bitwise AND where the result is a 1 if and only if both the bits on whom the AND operator is applied are also 1. In the above case, the third bits are 1 and the result is a 1, else the result is 0.

The Bitwise OR *|* is a little more lenient, if either of the bits is 1, the result is a 1. This happens when the bits line up as in positions 1 and 2. Obviously in the case of a OR, when both bits are 1, the result is also 1.

The XOR *^* is one of the most widely used bitwise operators in cryptography, it differs from the OR in only one way. If both bits are 1, then the result is a 0 and not a 1.

Therefore, the third bit is 0 and not 1 in a XOR computation. In non-cryptography cases, the Bitwise operations are used to set a certain bit to 1 or 0 or check whether a certain bit is a 1, etc.

In cryptography, i.e. in the case of hashes, the objective is to ascertain that the original bits to be hashed are changed in such a way that they appear random.

Simple Explanation for the Not Operator

```
ch1706.py
i = 5
k = ~i
print format(i & 0x000f, '04b')
```

```
print format(k & 0x000f, '04b')
```

Output

```
0101
1010
```

The NOT is the simplest Bitwise operator, it simply takes a 0 and makes it into a 1 and a 1 into a 0.

The & operator is used in the format function to mask out the last 12 bits. There is a 2's compliment as well but we will keep it for later.

An if Function for Bits and Not Bytes

ch1707.py

```
def ifch(b , c , d):
```

```
    f = ""
```

```
    b = format(b, '032b')
```

```
    c = format(c, '032b')
```

```
    d = format(d, '032b')
```

```
    i = 0
```

```
    print b
```

```
    print c
```

```
    print d
```

```
    for bi in b:
```

```
        bi = int(bi)
```

```
        if bi == 0:
```

```
            f = f + d[i]
```

```
        else:
```

```
            f = f + c[i]
```

```
        i = i + 1
```

```
    return int(f,2)
```

```
b = 0x67452301
```

```
c = 0x7bf36ae2
```

```
d = 0x98badcfe
```

```
f = ifch(b , c , d)
```

```
print format(f , '032b')
```

```
print
```

```
print "%08x" % f
```

```
f = d ^ (b & (c ^ d))
```

```
print "%08x" % f
```

```
f = (b & c) | ((~b) & d)
```

```
print "%08x" % f
```

Output

```
01100111010001010010001100000001
```

```
01111011111100110110101011100010
```

```
1001100010111010101110011111110
```

```

11111011111110111111111011111110
fbfbfefe
fbfbfefe
fbfbfefe

```

In this example, a bitwise if statement is used; an if statement for bits and not bytes.

Three variables *b*, *c* and *d* are created. These names match the names used in the SHA0 code. Next, these variables are initialized to values that will be calculated later.

Every line of code written is to explain the SHA hash copied from the open source sha hash source code, so nothing is randomly entered. Functions that behave like Bitwise if statements are also called Choice or IF functions.

The function *ifch* takes three parameters, *b*, *c* and *d*. These parameters first are converted into a binary string. This helps in working with every bit of the 4 bytes that make up the number.

Strings behave like arrays so it is easy to iterate every string bit and check for a zero or a one, one bit at a time. For each iteration of the loop, variable *bi* holds a bit from the parameter *b*. The first time in the loop, the bit *bi* will be 0 and the second time, the bit *bi* will be 1 as the second bit of the int *b* is 1. Read the displayed output starting from the left end, and not the right end to validate our claim. The variable *bi* is a string and not a number, so the *int* function is used to convert a string to a number, a standard Python practice.

Now a check is performed on the bit or the value of the variable *bi* to be 0 or 1. If it is 1, then the bit is true, so the corresponding bit from the next variable *c* is used. If the bit is 0 or false, the corresponding bit from the variable *d* is taken. The variable *i* is used which starts with 0 and is incremented by 1 at the end of the loop as an index to the strings *c* and *d*.

The variable *f* is an empty string in the beginning. Each time in the for loop, it would either pick up a bit string from the strings *c* or *d*. One string bit is concatenated to the string *f* each time in the loop.

The first bit in string *b* is a 0. Therefore, the first bit from string *d* is picked up which is 1. This is the why the first bit of the string *f* is 1. The second bit of the string *b* is 1. Therefore, bit 2 from the string *c* is picked up instead. The second bit of string *c* is 1 and the second bit of the string *f* is a 1. For the last time, the last bit of string *b* is a 1 and the last bit of string *c* is picked up which is a 0. This makes the last bit of string *f* a 0.

At the end of the function *ifch*, the process is reversed. The string *f* is converted into an integer using the same *int* function. Our string that represents an int can be in hex or decimal or binary, the default being decimal. In this case, the string contains either a 0 or 1, a binary string. Therefore, a value of 2 or binary is passed as the second parameter to the *int* function.

The original SHA specifications would not use functions like *ifch* in their sample code. They instead use the second and third form. The answers returned in each of the three cases are the same. We wrote this book to make things simpler. Once you understand our way, then use code written by the masters.

An Explanation of a Majority Function for Bits

```

ch1708.py
def maj(b , c, d):
    f = ""
    b = format(b, '032b')
    c = format(c, '032b')
    d = format(d, '032b')
    print b

```

```
print c
print d
i = 0
for bi in b:
    bi = int(bi)
    ci = int(c[i])
    di = int(d[i])
    tot1 = bi + ci + di
    tot0 = 3 - tot1
    if tot1 > tot0:
        f = f + '1'
    else:
        f = f + '0'
    i = i + 1
return int(f,2)

b = 0xac0c4120
c = 0x61c3d259
d = 0x8313b4ae

f = maj(b , c , d)
print format(f , '032b')
print
print "%08x" % f
f = (b & c) | (b & d) | (c & d)
print "%08x" % f
f = (b & c) ^ (b & d) ^ (c & d)
print "%08x" % f
```

Output

```
101011000000011000100000100100000
01100001110000111101001001011001
10000011000100111011010010101110
10100001000000111101000000101000

a103d028
a103d028
a103d028
```

Majority wins and that's what the maj function is assigned to do. The only difference is that the individual three bits of the string b, c and d are looked at. In the first bit, there are two 1's and one 0. The majority is a 1 and hence the result is a 1. For the second bit, there are 2 0's and only one 1. In this case, the 0's are in a majority and hence the result is a 0.

In the program code, the individual bits are converted to a number and then simply summed up. The value is saved in the tot1 variable. In effect, it gives a total number of 1's. A value of 3 is subtracted from the tot1 variable. The logic, though subtle is that there can be a maximum of only three 1's.

In the first case, we have two 1's, the tot0 variable has a value of 2. The tot0 has a value of 1 in the second case. Thus,

variable tot0 tells us the number of 0's present in the 3 strings at a certain position. The number of 0's cannot be ascertained directly as one cannot count the absence of a value.

If variable tot1 has a value larger than tot0 it means the 1's are in plenty. Any other logic can be applied to figure the same. The if statement determines if the 1's are in a majority or the 0's. A series of &'s and |'s do give the same answer.

A Parity Function for Bits

```
ch1709.py
def parity(b , c, d):
    f = ""
    b = format(b, '032b')
    c = format(c, '032b')
    d = format(d, '032b')
    print b
    print c
    print d
    i = 0
    for bi in b:
        bi = int(bi)
        ci = int(c[i])
        di = int(d[i])
        tot1 = bi + ci + di
        if tot1 %2 == 0:
            f = f + '0'
        else:
            f = f + '1'
        i = i + 1
    return int(f,2)
#f=15e9a74b
b = 0xc27d2218
c = 0x3eb579b7
d = 0xe921fce4
f = parity(b , c , d)
print format(f , '032b')
print
print "%08x" % f
f = f = b ^ c ^ d
print "%08x" % f
```

Output

```
11000010011111010010001000011000
00111110101101010111100110110111
1110100100100001111110011100100
00010101111010011010011101001011

15e9a74b
15e9a74b
```

Here, in the first bits there are 2 1's and the result is 0. In the 8th bit there is only one 1, the answer is 1. The result is 1 for an odd number of bits and 0 for an even number. If all the bits are 0, the answer is 0.

Getting the Data into a Format that the Sha Hash Wants

Output

Cryptographers like working with bits and not bytes. Here, the SHA hash is computed of the single character A which is 41 in hex. This character is only one byte long or takes up 8 bits. The SHA hash however requires the data to be hashed in chunks of 512 bits or 64 bytes, not higher nor lower. If the data is 65 bytes large then the SHA hash breaks it into two blocks of 64 bytes each.

The way out is to add the bit 1 at the end of the data. In our code, 0x80 is added as the binary bits for this value, it reads as 1000 0000. Remember one hex digit needs 4 binary bits to represent it. One hex digit is a number ranging from 0 to 15 or 16 unique numbers. This is the range of four binary bits.

The SHA hash likes the big-endian world. To avoid some security issues like a length extension attack, the length of the data, which in our case is 8 bits, is added at the end of the block. This length is however, added as an 8-byte big endian number. The 8 is added as the last byte of the 8 bytes and not the first. In the little-endian format, the number multiplied by one comes first whereas in big endian, this byte is last.

```
ch1711.py
m = "A" + '\x80' + '\x00' * 54 + "\x00\x00\x00\x00\x00\x00\x00\x08"
print m.encode('hex')
```

Output

The block of data is 512 bits large. Most of it is 0's but the underlying principles do not change. 512 bits divided by 32 gives 16 words or 16 numbers or 16 integers.

In the program, an array is created, or call it a list, which is 80 bytes large. In one step, this list is initialized to 0's. In a for loop, the variable `t` runs from 0 to 79 or a total of 80 rounds. All source code explaining the SHA hash uses such single character names, we do the same. In fact, all looping variables are a single character large. There is a certain poetic feel to writing code with one character variable names.

The first if statement with the condition `t <= 15` fills up the first 16 members of the list `w` with 32 bits from the original message. These values are integers but come across as hex digits. Using the array notation, 4 bytes are extracted at a time from the string `m` and then encoded. Only those bytes of the message are encoded that are used immediately. When `t` is 0, the slice operator become `0:4`, when `t` is 1, it is `4:8`, when `t` is 2 it is `8:12` etc., etc. This generates a string in hex, it's easy to create a number out of it.

303

Now comes the most important part of this loop. The task is to take some bits from the earlier 16 list members and spread them over the remaining 64 number list.

The SHA algorithm uses a simple method. It uses 4 of the previous offsets into the w array. They are offset 3,8,14 and 16. Remember, the variable t must be larger than or equal to 16 to enter the else statement. When the variable t has a value of 16, the offsets in the w array are 13,8, 2 and 0. The offsets value keep changing as the variable t keeps increasing. Finally, when the variable t has an ending value of 79, the offsets are 76, 71, 65 and 63 in the w array. It now goes beyond the initial 16 numbers.

We have no idea why these relative offsets were used. The designers did not stop here. They XOR'ed these four numbers and then stored the resultant value into list w with an offset of t larger than 16. The second series of numbers are displayed with a . to distinguish them from the first series.

In this manner, the SHA family fills up the other 64 numbers using the first 16 numbers of the original message. The idea again is to randomize and randomize the input data.

Shifting Bits to the Left and Right. A Left Rotate, the Right Way

```
ch1712.py
def _left_rotate(n, b):
    print "A " , format(n , "032b")
    print "B " , format(n << b , "032b") , len(format(n << b , "032b"))
    print "C " , format(n >> (32 - b) , "032b") , len(format(n >> (32 - b) , "032b"))
    print "D " , format((n << b) | (n >> (32 - b)) , "032b") , len(format((n << b) | (n >> (32 - b)) , "032b"))
    print "E " , format(((n << b) | (n >> (32 - b))) & 0xffffffff , "032b")
    return ((n << b) | (n >> (32 - b))) & 0xffffffff

shift = 1
n = 0x41800000
print format(n , "032b")
ns = n << shift
nss = format(ns , "032b")
print nss
print len(nss)
print

n = 0x83000000
print format(n , "032b")
ns = n << shift
nss = format(ns , "032b")
print nss
ns1 = n >> (32 - shift)
nss1 = format(ns1 , "032b")
print nss1
nss2 = format(ns | ns1 , "032b")
print "or " , nss2 , len(nss2)
nss3 = format((ns | ns1) & 0xffffffff , "032b")
print "and " , nss3 , len(nss3)
print len(nss)

print
```



```
a = _left_rotate(n , 1)
```

Output

```
01000001100000000000000000000000
10000011000000000000000000000000
32
10000011000000000000000000000000
10000011000000000000000000000000
00000000000000000000000000000001
or 10000011000000000000000000000001 33
and 000001100000000000000000000001 32
33
A 10000011000000000000000000000000
B 10000011000000000000000000000000 33
C 00000000000000000000000000000001 32
D 10000011000000000000000000000001 33
E 00000110000000000000000000000001
```

This is the last of the small examples before we dive deeply into SHA hash. Here we left shift the word that represents the message. It is very important for the security of the hash.

Rules for shifting of 1 or 20 remain the same. Right shifting by 2 is left shifting by 30. Python does not have a left shift operator which is circular. In a circular shift, the bits do not get thrown away, they re-appear from the other side.

Normal shifting is very easy, if the shift is 4 bits to the left, simply move all bits 4 to the left. Ditto for right. It is important to note that every time bits are shifted to the left by 1 the value is multiplied by 2. The binary bits have a weight 1, 2, 4, 8, 16 etc.

Let's take the first example where the value of n is 0x41800000. The hex representation is significant here. The first bit is 0, the second bit is 1 starting from the left moving right. When the bits are moved to the left, the 0 on the left falls off, that bit is now replaced by a 1, all the other bits move left by 1 and the incoming 0 is padded from the rear or right end.

The << is the non-circular Python left Bitwise operator. The problem is with a number like 0x83000000 where the first bit is 1. A left shift by 1 doesn't do much. No bits get shifted at all and a 0 get added at the other end or the extreme right. The length of the bits increases from 32 to 33.

Now let's right shift the variable ns by 31 bits, i.e. 32 - shift. This ensures that the offending 1, the top most bit is now at the other end but the length is now 32.

A basic principle of bitwise OR. When the operation is bitwise ORed with a 0, the bit being ORed remains the same. However, when bitwise ORing is with a 1, then the other bit must be 1. After right shifting, all the other bits are 0, so they do not make any difference. After bitwise ORing the above two, variables ns and ns1, a 33-bit number is obtained. Next task is to bitwise AND the same variables with all 1's. This operation is performed only on 32 bits. It removes the original offending 1 from the start.

A Bitwise AND with a 0 ensures that the resultant bit is 0 and a Bitwise AND with a 1 does not change the original bit value at all.

In the program, the int variable n is truncated to be a 32-bit int using a circular Bitwise left shift. The function `_left_rotate` is broken up into multiple print statements to simplify and ease the process. The entire number is first displayed in binary to see the original bits. This is A.

Then there is a left 'Python' shift by 1 as parameter b is 1. The original bits remain as they are but an extra 0 is added at the other end or the right. The number of bits increase by 1 to 33. This is B. Then there is a right shift 32 -1, i.e. 31 times. It moves the 1 to the extreme right and a 0 is added at the beginning. The two 1's in the middle also get eliminated. This is C. When there is a Bitwise OR on the above two bit strings, result is D and the length is 33. Finally, a Bitwise AND of 32 bits with a 1, leaves the 32 bits untouched and the value is in E.

In our code, we use the `_left_rotate` function to do the honors of a circular Bitwise left shift for us.

Computing a Sha0 Hash all by Ourselves

```
ch1713.py
def ifch(b , c , d):
    f = ""
    b = format(b, '032b')
    c = format(c, '032b')
    d = format(d, '032b')
    i = 0
    for bi in b:
        bi = int(bi)
        if bi == 0:
            f = f + d[i]
        else:
            f = f + c[i]
        i = i + 1
    return int(f,2)

def maj(b , c, d):
    f = ""
    b = format(b, '032b')
    c = format(c, '032b')
    d = format(d, '032b')
    i = 0
    for bi in b:
        bi = int(bi)
        ci = int(c[i])
        di = int(d[i])
        tot1 = bi + ci + di
        tot0 = 3 - tot1
        if tot1 > tot0:
            f = f + '1'
        else:
            f = f + '0'
        i = i + 1
    return int(f,2)

def parity(b , c, d):
    f = ""
    b = format(b, '032b')
    c = format(c, '032b')
```

```

d = format(d, '032b')
i = 0
for bi in b:
    bi = int(bi)
    ci = int(c[i])
    di = int(d[i])
    tot1 = bi + ci + di
    if tot1 % 2 == 0:
        f = f + '0'
    else:
        f = f + '1'
    i = i + 1
return int(f, 2)

def _left_rotate(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff

def sha1(m):
    w = [0] * 80
    a, b, c, d, e = (0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, 0xC3D2E1F0)
    for t in range(80):
        if t <= 15:
            w[t] = int(m[t*4:t*4 + 4].encode('hex'), 16)
        else:
            w[t] = w[t-3] ^ w[t-8] ^ w[t-14] ^ w[t-16]

        if t <= 19:
            f = d ^ (b & (c ^ d))
            f = (b & c) | ((~b) & d)
            f = ifch(b, c, d)
            k = 0x5A827999
        elif t <= 39:
            f = b ^ c ^ d
            f = parity(b, c, d)
            k = 0x6ED9EBA1
        elif t <= 59:
            f = (b & c) | (b & d) | (c & d)
            f = (b & c) ^ (b & d) ^ (c & d)
            f = maj(b, c, d)
            k = 0x8F1BBCDC
        elif t <= 79:
            f = b ^ c ^ d
            f = parity(b, c, d)
            k = 0xCA62C1D6

        T = (_left_rotate(a, 5) + f + e + k + w[t]) & 0xffffffff
        e = d
        d = c

```

```
c = _left_rotate(b, 30)
b = a
a = T

h0 = (0x67452301 + a) & 0xffffffff
h1 = (0xEFCDAB89 + b) & 0xffffffff
h2 = (0x98BADCFE + c) & 0xffffffff
h3 = (0x10325476 + d) & 0xffffffff
h4 = (0xC3D2E1F0 + e) & 0xffffffff
ans = '%08x%08x%08x%08x%08x' % (h0, h1, h2, h3, h4)
print ans
return ans

str = "A"
sha1(str + '\x80' + '\x00' * 54 + "\x00\x00\x00\x00\x00\x00\x00\x00\x08")

Output
e4da6a8fbd813c90e6fa040d5f15398eca200339
```

This example calculates the sha0 hash value for 'A'.

So, let's start with a quick revision. The function sha1 performs all the SHA hash computations. It is given the message, A. The value of A is hard coded as of now.

A list w is created with a size of 80 integers, all values as 0's. Then five variables called a, b, c, d, e are created, they are called chaining variables as their values will be reused in later blocks. Initially they are initialized to random values. We use the big-endian variants of the numbers that run from 0 to F and then backwards. The values are put in a tuple or list to save four lines of extra code.

The SHA0 hash algorithm runs all the data to be hashed through 80 rounds, which explains the for loop continuing 80 times. The loop variable is also called t. We could have extended our message from 16 words to 80 words by using a separate for loop, but we choose just one in our code, for no apparent reason.

The first part of the if statement is used to initialize the first 16 words of the list and the else is to spread out the original words. It remains a mystery as to why the block size was taken as 512 bits and not larger.

Things are a bit different now. The code for randomizing the bits for each round is not executed here. Instead, these rounds are further divided into 4 rounds of 20 each. Therefore, each of the if statements will execute 20 times depending upon the value of the loop variable, t.

The same task is performed in each of the four, second lot of if statements. A value of variable f is calculated and a constant value of k is chosen. This constant value k remains the same for 20 iterations. The values of k, is the square roots of 2, 3, 5 and 10 raised to 2 to the power of 30. As there are 4 loops of 20 each, four constants are needed. The variable f is simply calculated using the ifch function in the first if, the parity function in the second and fourth if and finally the maj function in the third.

After the if statements, the values of the variables a, b, c, d and e are changed. Thus, in every round, comparatively k will be the same (k will change only after 20 rounds) but the variable f will change every round as the values of variables b, c and d change. The variables a and e are not directly used in each round but influence the variables b, c and d to change.

At the end of every loop or round, at first a temporary variable T is calculated. This variable is basically a value that depends upon many variables. This is the most crucial variable of the lot. First, there is a circular left shift of the variable

a by 5 and then the newly calculated variable f is added to this value. To this value, e is added which has not been changed so far. The constant k has the same value for 20 iterations.

The most important value is the w or message list as it will change each time and its value represents the data to be hashed.

The number may become larger than a 32-bit number with multiple rotations or additions. So, always Bitwise AND with 8 f's to make sure it is 32 bits large only.

Now some more changes are introduced in the values of the 5 chaining variables. The variable e is set to d and then d to c. What's more, c is not changed to b but the variable is circular left rotated by 30. Variable b now takes on the value of variable a which in turn takes the value of T after b is initialized to the old a. We cannot explain why this happens.

In the next round, the value of f changes as the values of b, c and d change at the end of the loop. The value of e changes because the value of variable d changes, etc., etc.

There is only one round of 80 as the data is shorter than 448 bits. At the end of all the rounds, the same constants which were the running numbers 0123 etc. are added to the newly computed variables a to e. Once again, there is a check to not cross the 32-bit Rubicon.

The values in variables h0 to h5 when concatenated together become the final sha 160-bit hash value. To validate this SHA0 hash value, we look at the next program.

Double Checking that We have the Right Sha0 Hash

```
ch1714.py
import hashlib
def ifch(b , c , d):
    f = ""
    b = format(b, '032b')
    c = format(c, '032b')
    d = format(d, '032b')
    i = 0
    for bi in b:
        bi = int(bi)
        if bi == 0:
            f = f + d[i]
        else:
            f = f + c[i]
        i = i + 1
    return int(f,2)

def maj(b , c, d):
    f = ""
    b = format(b, '032b')
    c = format(c, '032b')
    d = format(d, '032b')
    i = 0
    for bi in b:
        bi = int(bi)
        ci = int(c[i])
        di = int(d[i])
```

```
    tot1 = bi + ci + di
    tot0 = 3 - tot1
    if tot1 > tot0:
        f = f + '1'
    else:
        f = f + '0'
    i = i + 1
    return int(f,2)

def parity(b , c, d):
    f = ""
    b = format(b, '032b')
    c = format(c, '032b')
    d = format(d, '032b')
    i = 0
    for bi in b:
        bi = int(bi)
        ci = int(c[i])
        di = int(d[i])
        tot1 = bi + ci + di
        if tot1 %2 == 0:
            f = f + '0'
        else:
            f = f + '1'
        i = i + 1
    return int(f,2)

def _left_rotate(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff

def sha1(m):
    w = [0] * 80
    a , b , c , d, e = (0x67452301, 0xEFCDAB89,0x98BADCFE ,0x10325476,0xC3D2E1F0)
    for t in range(80):
        if t <= 15:
            w[t] = int(m[t*4:t*4 + 4].encode('hex') , 16)
        else:
            w[t] = _left_rotate(w[t-3] ^ w[t-8] ^ w[t-14] ^ w[t-16], 1)
        if t <= 19:
            f = d ^ (b & (c ^ d))
            f = (b & c) | ((~b) & d)
            f = ifch(b , c , d)
            k = 0x5A827999
        elif t <= 39:
            f = b ^ c ^ d
            f = parity(b , c , d)
            k = 0x6ED9EBA1
```

```

elif t <= 59:
    f = (b & c) | (b & d) | (c & d)
    f = (b & c) ^ (b & d) ^ (c & d)
    f = maj(b, c, d)
    k = 0x8F1BBCDC
elif t <= 79:
    f = b ^ c ^ d
    f = parity(b, c, d)
    k = 0xCA62C1D6

T = (_left_rotate(a, 5) + f + e + k + w[t]) & 0xffffffff
e = d
d = c
c = _left_rotate(b, 30)
b = a
a = T

h0 = (0x67452301 + a) & 0xffffffff
h1 = (0xEFCDAB89 + b) & 0xffffffff
h2 = (0x98BADCFE + c) & 0xffffffff
h3 = (0x10325476 + d) & 0xffffffff
h4 = (0xC3D2E1F0 + e) & 0xffffffff
ans = '%08x%08x%08x%08x%08x' % (h0, h1, h2, h3, h4)
print ans
return ans

str = "A"
s = sha1(str + '\x80' + '\x00' * 54 + "\x00\x00\x00\x00\x00\x00\x00\x00\x08")
h = hashlib.sha1(str)
d = h.hexdigest()
if s == d:
    print "All is well"
else:
    print "Vijay Mukhi is an imbecile"

```

Output

```

6dcd4ce23d88e2ee9568ba546c007c63d9131c1b
All is well

```

Finally, we have taken some baby steps forward.

After NSA launched SHA, they quickly made one change to it and renamed it SHA0 and called the new hash algorithm SHA-1. NSA refuses to answer questions on why they dropped SHA-0. The only change they made was added a circular left rotation while computing the values of the last 64 numbers in the w array. This change makes the last 64 words of the computed message more random or so we understand.

In the program, this change is shown in the first else statement. The hashlib Python module brings in code to calculate the SHA-1 hash, it is similar to the code used for the SHA-256 hash. The function is called sha1.

Both the hash values are compared, the one we generated with the one computed by the module hashlib. The answer is the same. The hashlib module also has a function called sha1 which is given the string value, A. The same hexdigest

function gives a hex output of the hash value. This hash value we calculate, is stored in variable s and the hash computed by the hashlib module is stored in variable d.

This only proves that the program has calculated the right SHA hash value. The code in the module hashlib could be written in the C programming language for it to be faster, however the principles and constants remain the same.

Computing the Sha Hash on a Larger Message Size

```
ch1715.py
import hashlib
import struct
def _left_rotate(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff

def sha1(m):
    (h0 , h1 , h2 , h3 , h4) = ( 0x67452301 , 0xEFCDAB89 , 0x98BADCFE , 0x10325476 , 0xC3D2E1F0 )
    o = len(m)
    print "Length of m %d:%d" % (len(m),len(m) * 8)
    m += b'\x80'
    print "Length of m %d:%d" % (len(m),len(m) * 8)
    print "0's added %d" % ((o + 1))
    print "0's added %d" % (56 - (o + 1))
    print "0's added %d" % ((56 - (o + 1)) % 64)
    print "0's added %d" % ((56 - (o + 1) % 64) % 64)
    m += b'\x00' * ((56 - (o + 1) % 64) % 64)
    print "Length of m %d:%d" % (len(m),len(m) * 8)
    m += struct.pack(b'>Q', o * 8)
    print "Length of m %d:%d" % (len(m),len(m) * 8)
    print m.encode('hex')
    for i in range(0, len(m), 64):
        w = [0] * 80
        print "For Loop i=%d" % i
        for j in range(16):
            w[j] = struct.unpack(b'>I', m[i + j*4:i + j*4 + 4])[0]
        for j in range(16, 80):
            w[j] = _left_rotate(w[j-3] ^ w[j-8] ^ w[j-14] ^ w[j-16],1)
        (a , b , c , d , e) = (h0 , h1 , h2 , h3 , h4)
        for i in range(80):
            if 0 <= i <= 19:
                f = d ^ (b & (c ^ d))
                k = 0x5A827999
            elif 20 <= i <= 39:
                f = b ^ c ^ d
                k = 0x6ED9EBA1
            elif 40 <= i <= 59:
                f = (b & c) | (b & d) | (c & d)
                k = 0x8F1BBCDC
```



```

elif 60 <= i <= 79:
    f = b ^ c ^ d
    k = 0xCA62C1D6
    T = (_left_rotate(a, 5) + f + e + k + w[i]) & 0xffffffff
    e = d
    d = c
    c = _left_rotate(b, 30)
    b = a
    a = T

    h0 = (h0 + a) & 0xffffffff
    h1 = (h1 + b) & 0xffffffff
    h2 = (h2 + c) & 0xffffffff
    h3 = (h3 + d) & 0xffffffff
    h4 = (h4 + e) & 0xffffffff

    ans = '%08x%08x%08x%08x%08x' % (h0, h1, h2, h3, h4)
    print ans
    return ans

str = "My name is Vijay Mukhi and I am writing a book on CrtypAA"
s = sha1(str)
h = hashlib.sha1(str)
d = h.hexdigest()
if s == d:
    print "All is well"
else:
    print "Vijay Mukhi is an imbecile"

```

Output

```

Length of m 57:456
Length of m 58:464
0's added 58
0's added -2
0's added 62
0's added 62
Length of m 120:960
Length of m 128:1024
4d79206e616d652069732056696a6179204d756b686920616e64204920616d207772697
4696e67206120626f66b206f6e204372747970414180000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
For Loop i=0
For Loop i=64
b7cf63bc66428cc207b044bf3cf2f1cbdd3c37fd
All is well

```

We are finally squaring the circle. In this example, the SHA1 hash is calculated of data that exceeds 448 bits. This

complicates the code a lot. So, it makes more sense to break up the code, make many assumptions and then remove each assumption, one by one. We have adopted this strategy.

All eyes on the function sha1. At first, the size of our data both in bytes and bits is calculated. The variable o or parameter m has a size of 57 bytes or 456 bits at the start of the function sha1. This message has been chosen on purpose as the length must be larger than 448 bits, so that multiple messages are created. Ours is just a little bit larger. The string ends with two A's.

The bytes 01 c8 at the end of the second block or last block are the length of the string, c8 is $200 + (1 * 256)$ gives 456, the number of bits in the string. On adding one byte, a 0x80 to the message m will increase its length by 1 to 58 or its size to 464 bits. The output shows the two A's followed by a 0x80.

Any size exceeding 448 indicates one more block as the last 64 bits are to be used up for the length of all the data in bits.

Coming back to the data section, had it been exactly 448 bits in length, then it would be simple to add the length as the last 64 bits, $448 + 64 = 512$.

Here it is the entire message and not an individual message of 512 bits each. The size of the message after adding the 0x80 is 58 bytes. The variable o contains the original message size only and it is stored in parameter m. A byte, 0x80, is added to message m. As a result, the value of the variable is increased by 1, which explains the $o + 1$.

56 is subtracted from the total size of the message in bytes. The answer is -2 as the message size is larger than 56. It is 58. The message may be spread over 3 or more blocks. For this reason, we do a mod by 64 or 512 bits to get a multiple of 512 bits. Using mod ensures that the length is a certain multiple of a certain number 448 or 512 or anything. The magic number chosen is 512 as the length is added as the last 64 bits. Another mod operation is performed. Modding twice is purely for good luck, nothing else.

The last two mod's result in the same value of 62. All 0s are added at the end of the message m to give a message size of only 960 bits. The final big endian length is added to give a size of 1024 ($960 + 64$), a multiple of 512. Finally, a length of 1024 bits is achieved. The new message m is displayed in all its pristine glory. Again, first 0x80 is added at the end of the string m. Then the required number of 0's (to make it a multiple of 512 bits) are added after adding a 64-bit length at the end.

The code is further fine-tuned and a new for loop is introduced where the loop variable i increments in multiples of 64, as in 0, 64, 128 etc. The idea is to read 512 bits at a time into a list of 16 numbers. In the for loop, the list w is divided into 2 separate loops for no apparent reason. It has been separated from the main loop of 80 rounds only to make the code look logically correct. The only difference in the first for loop is that an offset of i is added to the slice operator. This loop will execute twice, with values of i being 0 and 64. As we are dealing with words, the value of i is increased by 64. Had it been bits the increase would be 512.

The messages are now read into the array w for every message block. In this specific case, the outer for loop runs twice as there are only two message blocks. Also, 5 new variables, h0 to h4 are created at the very start of the function and given random values. But the values in variable h0 to h4 will change later.

The chaining variables a to e are set to h0 to h4 as before. These values are set at the end of the for loop. So far, this initialization looks at par for the course. At the start of the second for loop, these chaining variables will reset their values.

Over the course of 80 rounds the first message block is handled. At the end of these 80 rounds, the random values in variables a to e change as well.

Before starting the second message block, the values in variables h0 to h5 are changed. The newly calculated values

of a to e are added. The values of variables a to e are changed 80 times in the second for statement, however, the variables h0 to h4 are changed once after exiting the inner for loop.

Now at the start of the second message block, the variables a to e are set not to the original values of h0 to h4, but to the changed values using the variables a to e. Therefore, variables a to e are called chaining variables as they chain themselves to the values computed using the previous message block. This is how randomness is introduced in the entire hash calculation.

The resultant calculated value will be different even if the second block of two messages are the same. The first block of the message block will give different values for variables a to e. And the previous block is used to influence the next block and so on. The final hash is the same as before, the values of the variables h0 to h4 concatenated together. The original sha1 hash also holds true. These values have been tested.

The Sha0 Hash is Broken Beyond Repair

```
ch1716.py
import struct
def _left_rotate(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff

def sha1(m):
    (h0 , h1 , h2 , h3 , h4) = ( 0x67452301 , 0xEFCDAB89 , 0x98BADCFE , 0x10325476 , 0xC3D2E1F0 )
    o = len(m)
    m += b'\x80'
    m += b'\x00' * ((56 - (o + 1) % 64) % 64)
    m += struct.pack(b'>Q', o * 8)
    for i in range(0, len(m), 64):
        w = [0] * 80

        for j in range(16):
            w[j] = struct.unpack(b'>I', m[i + j*4:i + j*4 + 4])[0]
        for j in range(16, 80):
            w[j] = w[j-3] ^ w[j-8] ^ w[j-14] ^ w[j-16]

        (a , b , c , d , e) = (h0 , h1 , h2 , h3 , h4)

        for i in range(80):
            if 0 <= i <= 19:
                f = d ^ (b & (c ^ d))
                k = 0x5A827999
            elif 20 <= i <= 39:
                f = b ^ c ^ d
                k = 0x6ED9EBA1
            elif 40 <= i <= 59:
                f = (b & c) | (b & d) | (c & d)
                k = 0x8F1BBCDC
            elif 60 <= i <= 79:
                f = b ^ c ^ d
                k = 0xCA62C1D6

            T = (_left_rotate(a, 5) + f + e + k + w[i]) & 0xffffffff
```

```
e = d
d = c
c = _left_rotate(b, 30)
b = a
a = T

h0 = (h0 + a) & 0xffffffff
h1 = (h1 + b) & 0xffffffff
h2 = (h2 + c) & 0xffffffff
h3 = (h3 + d) & 0xffffffff
h4 = (h4 + e) & 0xffffffff

ans = '%08x%08x%08x%08x' % (h0, h1, h2, h3, h4)
return ans

d =
'a766a602b65cffe773bcf25826b322b3d01b1a972684ef533e3b4b7f53fe376224c08e47
e959b2bc3b519880b9286568247d110f70f5c5e2b4590ca3f55f52feeffd4c8fe68de8353
29e603cc51e7f02545410d1671d108df5a4000dcf20a4394949d72cd14fbb0345cf3a295d
cda89f998f87552c9a58b1bdc384835e477185f96e68bebb0025d2d2b69edf21724198f6
88b41deb9b4913fbe696b5457ab39921e1d7591f89de8457e8613c6c9e3b242879d4d87
83b2d9ca9935ea526a729c06edfc50137e69330be976012cc5dfe1c14c4c68bd1db3ecb2
4438a59a09b5db435563e0d8bdf572f77b53065cef31f32dc9dbaa04146261e9994bd5cd
0758e3d'.decode('hex')
ans = sha1(d)
print ans
d1 =
'a766a602b65cffe773bcf25826b322b1d01b1ad72684ef51be3b4b7fd3fe3762a4c08e45
e959b2fc3b51988039286528a47d110d70f5c5e034590ce3755f52fc6ffd4c8d668de8753
29e603e451e7f02d45410d1e71d108df5a4000dcf20a4394949d72cd14fbb0145cf3a695
dcda89d198f8755ac9a58b13dc384815e4771c5796e68febb0025d052b69edda17241d8
7688b41f6b9b49117be696f5c57ab399a1e1d7199f89de8657e8613cec9e3b26a879d498
783b2d9e29935ea7a6a729806edfc50337e693303e9760104c5dfe5c14c4c68951db3ecb
a4438a59209b5db435563e0d8bdf572f77b53065cef31f30dc9dbae04146261c1994bd5c
50758e3d'.decode('hex')
ans1 = sha1(d1)
print ans1
if ans == ans1:
    print "The SHA 0 hash is broken"
else:
    print "The SHA0 hashes are different"
if d == d1:
    print "Strings Equal"
else:
    print "Strings d and d1 are not the same"
```

Output

```
c9f160777d4086fe8095fba58b7e20c228a4006b
```

c9f160777d4086fe8095fba58b7e20c228a4006b
 The SHA 0 hash is broken
 Strings d and d1 are not the same

This example explains the reasons behind the decline of the SHA-0 hash. It is the same example as before, but no circular left shifting of bits by 1. The SHA-0 hash of two data pieces are calculated. It comes as no surprise that the hashes are the same but the strings are different. The output shows that for two different strings, the same sha hash is displayed, the security is compromised. One more confirmation that our code works.

This shows that collisions have been found in the SHA-0 hash and therefore the NSA abandoned their own prodigy.

Why Random (Nothing Up Your Sleeve) Numbers are Important

```
ch1717.py
def _left_rotate(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff

def sha1(m):
    w = [0] * 80
    a, b, c, d, e = (0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, 0xC3D2E1F0)
    for t in range(80):
        if t <= 15:
            w[t] = int(m[t*4:t*4 + 4].encode('hex'), 16)
        else:
            w[t] = _left_rotate(w[t-3] ^ w[t-8] ^ w[t-14] ^ w[t-16], 1)
        if t <= 19:
            f = d ^ (b & (c ^ d))
            k = 0x5a827999
        elif t <= 39:
            f = b ^ c ^ d
            k = 0x4eb9d7f7
        elif t <= 59:
            f = (b & c) | (b & d) | (c & d)
            k = 0xbad18e2f
        elif t <= 79:
            f = b ^ c ^ d
            k = 0xd79e5877

        T = (_left_rotate(a, 5) + f + e + k + w[t]) & 0xffffffff
        e = d
        d = c
        c = _left_rotate(b, 30)
        b = a
        a = T

    h0 = (0x67452301 + a) & 0xffffffff
    h1 = (0xEFCDAB89 + b) & 0xffffffff
    h2 = (0x98BADCFE + c) & 0xffffffff
    h3 = (0x10325476 + d) & 0xffffffff
```

```
h4 = (0xC3D2E1F0 + e) & 0xffffffff
ans = '%08x%08x%08x%08x' % (h0, h1, h2, h3, h4)
return ans

d =
"fffd8ffe1e2001250b6cef60834f4fe83ffae884fafe56e6ffc50fae628c40f811b1d3283b48
c11bcb1d4b511a976cb20a7a929f02327f9bbecde01c07dc00852".decode('hex')
ans = sha1(d)
d1 =
"fffd8ffe2c20012243ecef608dcf4fee137ae880c87e56e6bbc50faa460c40fc7931d3281b48
c11a8b9d4b5130976cb742fa929f2a327f9bb44de01c3d5c00832".decode('hex')
ans1 = sha1(d1)
print ans
print ans1
if ans == ans1:
    print "The SHA 1 hash is broken"
else:
    print "The SHA1 hashes are different"
if d == d1:
    print "Strings Equal"
else:
    print "Strings d and d1 are not the same"
```

Output

```
b3cff78b4d461ace4d0a8f9cfc21246fb2581090
b3cff78b4d461ace4d0a8f9cfc21246fb2581090
The SHA 1 hash is broken
Strings d and d1 are not the same
```

This example claims that the SHA-1 hash has been broken. Well Yes and No. Let's start with the No.

There are two different sets of data in variables d and d1 which vary slightly. The hash value computed however, is the same. The program on the face of it, looks the same.

There are four different constants called k used in the 80 rounds of statement. The value of the second constant is 0x4eb9d7f7, the original was 0x6ED9EBA1. In the same vein the third constant k is 0xbad18e2f, earlier it was 0x8F1BBCDC.

All that this example proves is that if the wrong values of the constants k are chosen, then there will be a collision with the SHA-1 hash. Remember these data sets have been specially crafted for the given constants.

The hashlib library is not used for computing the hash as they will not use the compromised constants given in the code.

Cryptography must be open source and for all constants, random numbers must be used. If not, then a backdoor or a trapdoor will be used to decrease the security of the hash. A certain set of constants that look very innocuous may allow the designer of the hash to get away with murder.

Please visit the website <https://malicioussha1.github.io> from where this code is borrowed. They have a zillion more examples on how security can be compromised. A big thank you to the person who wrote this code. We should have complimented by name, everyone who inspired us.

Is the SHA-1 hash broken? No, but as the above website explains, lots of implementers tweak the SHA hash in small ways which allow attackers to compromise the hash.

Making the Sha Hash Use Less Memory

```
ch1718.py
import hashlib
def _left_rotate(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff

def sha1(m):
    w = [0] * 16
    a, b, c, d, e = (0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, 0xC3D2E1F0)
    MASK = 0x0000000f
    for t in range(80):
        s = t & MASK
        print s,
        if t <= 15:
            w[t] = int(m[t*4:t*4 + 4].encode('hex'), 16)
        else:
            w[s] = _left_rotate(w[(s+13 & MASK)] ^ w[(s+8) & MASK] ^ w[(s+2) & MASK] ^ w[s], 1)
        if t <= 19:
            f = d ^ (b & (c ^ d))
            k = 0x5A827999
        elif t <= 39:
            f = b ^ c ^ d
            k = 0x6ED9EBA1
        elif t <= 59:
            f = (b & c) | (b & d) | (c & d)
            k = 0x8F1BBCDC
        elif t <= 79:
            f = b ^ c ^ d
            k = 0xCA62C1D6
        T = (_left_rotate(a, 5) + f + e + k + w[s]) & 0xffffffff
        e = d
        d = c
        c = _left_rotate(b, 30)
        b = a
        a = T
    h0 = (0x67452301 + a) & 0xffffffff
    h1 = (0xEFCDAB89 + b) & 0xffffffff
    h2 = (0x98BADCFE + c) & 0xffffffff
    h3 = (0x10325476 + d) & 0xffffffff
    h4 = (0xC3D2E1F0 + e) & 0xffffffff
    ans = '%08x%08x%08x%08x%08x' % (h0, h1, h2, h3, h4)
    print "\n%s" % ans
```

```
    return ans

str = "A"
s = sha1(str + '\x80' + '\x00' * 54 + "\x00\x00\x00\x00\x00\x00\x00\x00\x08")
h = hashlib.sha1(str)
d = h.hexdigest()
if s == d:
    print "All is well"
else:
    print "Vijay Mukhi is an imbecile"
```

Output

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5
6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10
11 12 13 14 15
6dcd4ce23d88e2ee9568ba546c007c63d9131c1b
All is well
```

In our journey of understanding hashes, we came across a program in the specifications that tried to optimize memory usage.

In the 1990s, creating a list or arrays that could hold 80 numbers was only for the rich and famous and the high and mighty. The SHA-1 hash calculations had to run on embedded small devices. The powers to be decided that having a list 80 large is good for speed but bad for memory. They were right. In the above example, we reduce the size of the list `w` to only 16 numbers and use it as a circular buffer.

Initially the list `w` is created and it is only 16 members large. There is a constant `MASK` that has a value of decimal 15 or a `F`. As explained before, Bitwise ANDing with a 1 does not change a bit but a Bitwise AND with a 0, will give a 0 irrespective of the other bit. In this case, the value of `s` is guaranteed to be a number between 0 to 15, whatever happens. The output confirms what we are saying. For the first 16 numbers, the same routine process is applied but for the rest of the numbers larger than 16, something remarkably different is attempted.

The list index is `s` and the list is only 16 large. Variable `s` cannot be larger than 15 and it circles from 0 to 15. In the `else` block, the order is different but the list indexes are the same 13 8 2 and 0, when variable `t` has a value of 16. The trick is in how the actual elements of the list `w` are computed with the `MASK` each time. This ensures that other members of the list are rightly referenced. Try it out by hand to understand, our explanations will not help. Can and should be ignored, a simple optimization. That's why highly optimized code is difficult to explain. Many ways to skin a cat.

CHAPTER 18

Hashes - Sha-256 and RipeMD-160

The chapter starts with an in-depth understanding of the Sha-256 hash algorithm and shows the differences from the Sha-1 hash. The Sha-256 hash gives a 256-bit hash value. A 160-bit hash requires five 32 bit numbers, a 256-bit hash requires eight 32 bit numbers. Therefore, instead of five, eight chaining variables are used as in a, b, c, d, e, f, g and h. These variables are initialized to some random numbers.

The Eight Chaining Variables Used in Sha-256

```
ch1801.py
import math
print hex(int(math.modf(math.sqrt(2))[0]*(1<<32)))
print math.sqrt(2)
print int(math.sqrt(2))
print math.modf(math.sqrt(2))
print math.modf(math.sqrt(2))[0]
print math.modf(math.sqrt(2))[0] * (1 << 32)
print int(math.modf(math.sqrt(2))[0] * (1 << 32))
print hex(int(math.modf(math.sqrt(2))[0] * (1 << 32)))
print

print '0x'+math.sqrt(2).hex().split('.')[1][:8]
print math.sqrt(2).hex()
print math.sqrt(2).hex().split('.')[1]
print math.sqrt(2).hex().split('.')[1]
print math.sqrt(2).hex().split('.')[1][:8]
print '0x' + math.sqrt(2).hex().split('.')[1][:8]
print

print hex(int(math.modf(math.sqrt(3))[0]*(1<<32)))
print hex(int(math.modf(math.sqrt(5))[0]*(1<<32)))
print hex(int(math.modf(math.sqrt(7))[0]*(1<<32)))
print hex(int(math.modf(math.sqrt(11))[0]*(1<<32)))
print hex(int(math.modf(math.sqrt(13))[0]*(1<<32)))
print hex(int(math.modf(math.sqrt(17))[0]*(1<<32)))
print hex(int(math.modf(math.sqrt(19))[0]*(1<<32)))
```

Output

```
0x6a09e667
1.41421356237
```

```
1
(0.41421356237309515, 1.0)
0.414213562373
1779033703.95
1779033703
0x6a09e667

0x6a09e667
0x1.6a09e667f3bcdp+0
['0x1', '6a09e667f3bcdp+0']
6a09e667f3bcdp+0
6a09e667
0x6a09e667

0xbb67ae85
0x3c6ef372
0xa54ff53a
0x510e527f
0x9b05688c
0x1f83d9ab
0x5be0cd19
```

The Sha-256 hash is better documented than the Sha-1 hash in the FIPS-180-4 standard. The eight chaining variables are initialized to the square root of the first 8 prime numbers which are 3,5,7,11,13,17 and 19. Square roots gives numbers with decimal places and the fractional parts of these numbers are also used.

The code can get very confusing so, it's been broken up into smaller fragments. The square root of 2 is 1.4142..... Taking an int of this number with decimal places would give an integer which is a 1. We want the fractional part of this value also.

The modf function from the math module is just what the doctor ordered. This function breaks up the both the parts at the decimal point and returns a tuple.

The FIPS standard looks at the fractional part, so the first value in the tuple, the [0] is used. There is a number with decimal places, so the bits are left shifted by 32 which in effect is a multiplication by 2 raised to 32, it removes the all the decimal places.

The int function removes the decimal parts. As the number is to be displayed as a hex value and not decimal, the hex function is used to re-check that the constant has the right values.

There are multiple ways to skin a cat in programming languages. An easier way is to first use the hex function to convert the square root value into a hex number. It displays an actual hex string starting with a 0x (we place the 0x). The decimal point onwards is the hex value we are looking for. It is a string. The required values are after the decimal point, so the split function is used on the string at the decimal point. The net result is that there are two sets, the real integer part and the fractional part.

The array notation [1] on the tuple gives only the fractional part and the slice operator [:8] gives the first 8 digits. The 0x is added only for effect.

The rest of the code performs the routine task, one step at a time. The other seven chaining variables are also displayed.

64 Constants that are the Cube Root of the First 64 Prime Numbers

```
ch1802.py
import math
p = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149,
151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311]
print "Number of primes are %d" % len(p)
for i in p:
    print "%sL," % hex(int(math.modf(i**(1/3.0))[0] * (1 << 32))) ,
```

Output

```
Number of primes are 64
0x428a2f98L, 0x71374491L, 0xb5c0fbcfL, 0xe9b5dba5L, 0x3956c25bL, 0x59f111f1L,
0x923f82a4L, 0xab1c5ed5L, 0xd807aa98L, 0x12835b01L, 0x243185beL,
0x550c7dc3L, 0x72be5d74L, 0x80deb1feL, 0x9bdc06a7L, 0xc19bf174L,
0xe49b69c1L, 0xefbe4786L, 0xfc19dc6L, 0x240ca1ccL, 0x2de92c6fL, 0x4a7484aaL,
0x5cb0a9dcL, 0x76f988daL, 0x983e5152L, 0xa831c66dL, 0xb00327c8L, 0xbf597fc7L,
0xc6e00bf3L, 0xd5a79147L, 0x6ca6351L, 0x14292967L, 0x27b70a85L, 0x2e1b2138L,
0x4d2c6dfcL, 0x53380d13L, 0x650a7354L, 0x766a0abbL, 0x81c2c92eL,
0x92722c85L, 0xa2bfe8a1L, 0xa81a664bL, 0xc24b8b70L, 0xc76c51a3L,
0xd192e819L, 0xd6990624L, 0xf40e3585L, 0x106aa070L, 0x19a4c116L,
0x1e376c08L, 0x2748774cL, 0x34b0bcb5L, 0x391c0cb3L, 0x4ed8aa4aL,
0x5b9cca4fL, 0x682e6ff3L, 0x748f82eeL, 0x78a5636fL, 0x84c87814L, 0x8cc70208L,
0x90befffaL, 0xa4506cebL, 0xbef9a3f7L, 0xc67178f2L
```

In the Sha-1 hash, there are 80 rounds where the same constant is used for 20 rounds. Therefore, only 4 constants are needed. As seen in the earlier program, changing the constants could lead to the net hashes colliding with each other.

In Sha-256, the number of rounds are reduced from 80 to 64. Also in each round, there is a new constant which is the cube root of the first 32 bits of the fractional part of the first 64 prime numbers. That is quite a mouthful.

The FIPS, gives these values but we prefer to create our own. The array p contains the first 64 prime numbers (using Google). In the last century, we used Fortran to generate these prime numbers.

In a for loop, the cube root of a number is acquired by looping through the prime number array p and using the ** operator. One of the parameters, 1 or 3 must be a floating-point number in python version 2.7 or lower, but not in Python 3.0. Then, the same code is used to extract the factional part. Please Check the FIPS standard for the recommended constants.

Using the Zip Function to Add Lists

```
ch1803.py
s1 = ""
for x,y in zip([1,2,3], [4,5,6]):
    print x,y
    s1 = s1 + "%08x" % ((x+y) & 0xFFFFFFFF)
print s1
```

Output

```
1 4
2 5
3 6
000000050000000700000009
```

Imagine a situation, where there are two lists of 8 numbers and they are to be added using the same offsets. One way is to use eight separate statements or use the zip function with the two lists and have a for loop iterate through them.

In our code, there are three members in the list, so the for loop iterates three times. The x and y are the individual members of the array having the same index or offset position.

In the for loop, the two members of the list are added. Next the AND operator ensures that the value is not beyond a 32-bit number. Finally, the value is converted into a hex string and concatenated to s1. This hex string in s1 is the 256-bit hash value.

Rotate Bits Left or to the Right

```
ch1804.py
def _rotr(x, y):
    return ((x >> y) | (x << (32-y))) & 0xFFFFFFFFL

def _left_rotate(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff

a = 0x81
b = _rotr(a, 3)
print "%x" % b
b = _left_rotate(a, 32 - 3)
print "%x" % b
```

Output

```
20000010
20000010
```

This program needs no explanation but we will write something.

The program starts with a number in variable a which is 0x81. In the first case, there is a rotate right by 3, which is the equivalent of rotating left by 29. The user-defined functions used are _rotr and _left_rotate. These functions have been used before in some previous chapter.

The result is the same. For some reason the Sha-256 world prefers rotating right and not rotating left.

Computing the Sha-256 Hash and Checking the Hash with a Standard Library

```
ch1805.py
import struct
import hashlib
k = (0x428a2f98L, 0x71374491L, 0xb5c0fbcfL, 0xe9b5dba5L, 0x3956c25bL,
0x59f111f1L, 0x923f82a4L, 0xab1c5ed5L, 0xd807aa98L, 0x12835b01L, 0x243185beL,
0x550c7dc3L, 0x72be5d74L, 0x80deb1feL, 0x9bdc06a7L, 0xc19bf174L, 0xe49b69c1L,
0xefbe4786L, 0x0fc19dc6L, 0x240ca1ccL, 0x2de92c6fL, 0x4a7484aaL, 0x5cb0a9dcL,
```

[illegible]

```
def sha256():
    w = [0]*64

    w[0:15] = struct.unpack('>16L', m)

    for i in range(16, 64):
        s0 = _rotr(w[i-15], 7) ^ _rotr(w[i-15], 18) ^ (w[i-15] >> 3)
        s1 = _rotr(w[i-2], 17) ^ _rotr(w[i-2], 19) ^ (w[i-2] >> 10)
        w[i] = (w[i-16] + s0 + w[i-7] + s1) & 0xFFFFFFFFFL

    a,b,c,d,e,f,g,h = _h

    for i in range(64):
        s0 = _rotr(a, 2) ^ _rotr(a, 13) ^ _rotr(a, 22)
        maj = (a & b) ^ (a & c) ^ (b & c)
        t2 = s0 + maj

        s1 = _rotr(e, 6) ^ _rotr(e, 11) ^ _rotr(e, 25)
        ch = (e & f) ^ ((~e) & g)
        t1 = h + s1 + ch + k[i] + w[i]

        h = g
        g = f
        f = e
        e = (d + t1) & 0xFFFFFFFFFL
        d = c
        c = b
        b = a
        a = (t1 + t2) & 0xFFFFFFFFFL

    s1 = ''
    for x,y in zip(_h, [a,b,c,d,e,f,g,h]):
        s1 = s1 + "%08x" % ((x+y) & 0xFFFFFFFF)
    print s1
```

```
        return s1
a = sha256()
b = hashlib.sha256('A').hexdigest()
if a == b:
    print "All is Good"
else:
    print "Vijay Mukhi is an imbecile"
```

Output

```
559aead08264d5795d3909718cdd05abd49572e84fe55590eef31a88a08fdffd
All is Good
```

This example calculates the Sha-256 hash value. Plus, the output is verified with the functions from the hashlib library. This code is cleaner and looks simpler.

The Sha-256 hash algorithm has 64 rounds and a separate constant is required for each of these rounds. The list k is the cube root of the fractional part of the first 64 prime numbers. The temp variable t1 at the end of each round is calculated using this list k[i].

The message m is made up of the string value A followed by the 1 bit followed by 448 0's and then the big-endian length. All hashes store the message in this format.

The final hash created is 256 bits or $32 * 8$ or 8 integers or words. At first, these eight chaining variables a, b, c, d, e, f, g and h are initialized to some value. This value is the square root of the fractional part of the first 8 prime numbers. This list is called `_h` because the original source code on the internet uses `_h`. In place of the circular rotate left function, the `_rotr` or rotate right is used which rotates bits to the right.

Now for the real code in the function sha256. A list w with a length of 64 is created as there are 64 rounds and this list is initialized to all 0's. Like before, the first 16 numbers of the list must be initialized to the 512 bits of the message. In place of loops, the unpack function is used which unpacks 16 numbers from the 512 bit or 16 int messages, m and returns 16 values. These values are stored in the first 16 members of the w list. More compact code.

Now, to initialize the other 48 members of the list w. Instead of one left rotate, the designers bought in 4 circular right rotates and 2 non-circular right rotates. The Sha-256 designers saw how one left shift killed Sha-0 so, they decided to hit back with vengeance by going overboard.

The values are stored in temp variables s0 and s1 otherwise all the operations will not fit on one screen. The offsets of the previous message used are 15, 2, 16 and 7, all in a random order. The variables, s1 and s2 are tmp variables.

For the first iteration when i is 16, the offsets used are 1, 14, 0 and 9. This is the single most important change made to the Sha-256 hash, the message bits are fully randomized. The chaining variables a to h are set as IV's to the `_h` list.

The second for loop could be merged with the first but it's been kept separate for simplicity and readability. The two parity functions are also eliminated. The same Choice or ch or Majority or maj functions are maintained for each round. The other difference is that the maj functions uses the chaining variables a, b and c. The Choice function uses e, f and g. They divide up the chaining variables.

For some reason, the sha256 has a fetish for rotation. The variable a is first rotated with three different values, which are 2 and 13 and 22. For this purpose, a temp variable t2 is created which is a sum of the majority function and the multiple rotations of variable a.

Now things get more complicated. The e chaining variable is rotated 3 times, 6 and 11 and 25. The temp variable t1 is used for storing many things. It is the sum of the chaining variable h that is not used so far. Then there is the rotation of

by the other hashes. The code, we had access to, preferred computing the numbers using the unpack function instead of simply laying them out. So, no arguments here.

In RipeMD, the 512-bit string is stored as a list of sixteen 32 bit numbers. These 512 bits are not extended further over 64 numbers. Thus, in RipeMD, the size of array w of words is kept to its original size. No extensions at all.

Now things change. Instead of having a single for loop of 80 rounds, there are 2 main rounds that run in parallel. Each of them, call the box function 5 times. The box function iterates 16 times. Each box function returns a tuple of size 5 which means that, each for statement iterates 80 times, twice, side by side.

Let's now understand the box function and the values given to it during its iterations.

Function Programming, Understanding a Lambda

ch1807.py

```
f1 = lambda x, y, z: x + 10
f2 = lambda x, y, z: y + 2 * z
fl = [f1, f2]
print type(f1)
print fl[0](1,2,3)
print fl[1](1,2,3)
```

Output

```
<type 'function'>
11
8
```

Functional programming languages treat functions as first class citizens. A variable can represent a function. In Python, there is a keyword called lambda to accomplish the same.

The variable f1 has a type of function as the keyword lambda is used. This is followed by the parameters to the function, in this case variables x y and z. Then comes a colon followed by the body of the function. Ditto for variable f2.

Next, a list is created called fl comprising of two members whose members are functions, f1 and f2. A list can have any data type in Python. To call a function, we simply access the member of the list as fl[0] or fl[1] and then pass the parameters in () brackets.

The program basically depicts that a list can have function types as well.

Finding Square and Cube Roots the Lambda Way

ch1808.py

```
_kg = lambda x, y: int(2**30 * (y ** (1.0 / x)))
KL = [0, _kg(2, 2), _kg(2, 3)]
KR = [_kg(3, 2), _kg(3, 3), _kg(3, 5)]

print "%08x" % _kg(2, 3)
print "%08x" % int(2**30 * (3 ** (1.0 / 2)))
print "%08x" % KL[2]

print "%08x" % _kg(3, 5)
print "%08x" % int(2**30 * (5 ** (1.0 / 3)))
print "%08x" % KR[2]
```


Output

```
6ed9eba1
6ed9eba1
6ed9eba1
6d703ef3
6d703ef3
6d703ef3
```

There was the square root and cube root of numbers in the programs of the previous chapters. They were then multiplied by 2 raised to 30. For example, the constant 6ed9eba1 was the square root of 3 multiplied by 2 raised to 30.

In place of a single method for calculating this value, the program uses a variable called `_kg` which is of function type, thanks to the lambda keyword. This function requires two parameters. The second parameter `y` is raised to the first. If the parameter `x` is 2, then the operation is a square root, if 3 then a cube root.

A gentle reminder that the decimal number 1.0 is to force Python 2.7 to use floating point numbers. Also, the square or cube root is to be multiplied by 2 raised to 30 and then the int of the result is taken.

The list KL calls the square root of the second parameter passed, as the first parameter is 2. The list KR finds the cube root of numbers 2, 3 and 5.

All the three print commands display the same value. In the first case, the function `_kg` is called directly. The second parameter 3 is raised to the power of 2. In the next print statement, the same value is calculated by the raised to and the multiplication directly. Finally, in the third case, the last member of the list KL is called which in turn calls the `_kg` function with the same parameters as in the first print. In the second set of print commands, the cube root value is determined as the value is 3 instead of 2. The values are calculated dynamically.

Misery with Python Lists within Lists

```
ch1809.py
rho = [7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]
print "rho=%s" % rho
rl = [range(16)]
print "\nrl[-1] and rl[0] values are the same"
print "rl=%s" % rl
print "rl[-1]=%s" % rl[-1]
print "rl[0]=%s" % rl[0]
rl += [[rho[j] for j in rl[-1]]]
print "\nrl[-1] and rl[0] values after the first addition"
print "rl=%s" % rl
print "rl[-1]=%s" % rl[-1]
print "rl[0]=%s" % rl[0]
rl += [[rho[j] for j in rl[-1]]]
print "\nrl[-1] and rl[0] values after the 2nd addition. 3 arrays"
print "rl=%s" % rl
print "rl[-1]=%s" % rl[-1]
print "rl[-2]=%s" % rl[-2]
print "rl[-3]=%s" % rl[-3]
print "rl[0]=%s" % rl[0]
print "rl[1]=%s" % rl[1]
```

```
print "rl[2]=%s" % rl[2]
rl += [[rho[j] for j in rl[-1]]]
print "\nrl[-1] and rl[0] values after the 3rd addition. 4 arrays"
print "rl=%s" % rl
print "rl[-1]=%s" % rl[-1]
rl += [[rho[j] for j in rl[-1]]]
print "\nrl[-1] and rl[0] values after the last addition. 5 arrays"
print "rl=%s" % rl
print "rl[-1]=%s" % rl[-1]
print
```

Output

```
rho=[7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]

rl[-1] and rl[0] values are the same
rl=[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]]
rl[-1]=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
rl[0]=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

rl[-1] and rl[0] values after the first addition
rl=[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], [7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]]
rl[-1]=[7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]
rl[0]=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

rl[-1] and rl[0] values after the 2nd addition. 3 arrays
rl=[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], [7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8], [3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12]]
rl[-1]=[3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12]
rl[-2]=[7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]
rl[-3]=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
rl[0]=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
rl[1]=[7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]
rl[2]=[3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12]

rl[-1] and rl[0] values after the 3rd addition. 4 arrays
rl=[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], [7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8], [3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12], [1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2]]
rl[-1]=[1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2]

rl[-1] and rl[0] values after the last addition. 5 arrays
rl=[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], [7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8], [3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12], [1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2], [4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8, 11, 6, 15, 13]]
rl[-1]=[4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8, 11, 6, 15, 13]
```

RipeMD makes it very difficult for a hacker to break the security of its hash. It also makes our life difficult to explain this code. Let's look at what the doctor ordered.

There is a list, rho that has 16 numbers ranging from 0 to 15, which eventually refers to the 16 members of the message. These numbers are randomly ordered. The RipeMD family does everything in steps of 16.

A list, rl is created using the range function. The range function returns numbers from 0 to 15. A careful observation shows that the rl list has only one element, which is a list of 16 elements. The beauty of lists is that an offset of -1 gives the last element of the list. As the list has only one element, -1 and 0 gives the same list, as displayed in the output.

Now to the tougher part. The rl[-1] is the list that runs from 0 to 15. It is used to access the rho list. You get the same list as the rl[-1] is an ordered list. This is because the rho list is copied as the last item in the rl list of lists. Now the list has two elements.

Now comes the deal breaker. The rl[-1] list is 7, 4, 13 etc. as it is the last list. The seventh member of the rho list is 3, the fourth member is 10 and the 13 member is 14. The new list added is now 3, 10, 14 etc. In all, there are 3 elements in our list. So, we have created lists within lists with 16 members and they contain numbers in the range of 0 to 15. A simpler way would be to perform these calculations of lists outside and use them in the actual lists.

Simplifying the Lists within Lists

```
ch1810.py
rho = [7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]
rl = [range(16)]
rl += [[rho[j] for j in rl[-1]]]
print rl
_shift1 = [11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8]
_shift2 = [12, 13, 11, 15, 6, 9, 9, 7, 12, 15, 11, 13, 7, 8, 7, 7]
sl = [[_shift1[rl[0][i]] for i in range(16)]]
print sl
sl.append([_shift2[rl[1][i]] for i in range(16)])
print sl
print sl[0]
```

Output

```
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], [7, 4, 13, 1, 10, 6, 15, 3, 12,
0, 9, 5, 2, 14, 11, 8]]
[[11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8]]
[[11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8], [7, 6, 8, 13, 11, 9, 7, 15, 7,
12, 15, 9, 11, 7, 13, 12]]
[11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8]
```

The earlier program was confusing, so we are making it simpler. The same list rho is initialized with values and then a list of 2 lists is created, one with simple numbers from 0 to 15 and second with the members of rho. The two lists can be seen in the rl list.

There are 2 arrays, _shift1 and _shift2 with some more random numbers from 0 to 15, a total of 16 as always. There are duplicates like 9.

A new list, sl is created in the following way. The first time around, variable i has a value of 0. The variable rl[0] is the first list and [0] (because of variable i) from it will give a value of 0. Continuing further, there will be a series of numbers from 0 to 15 and the _shift1 will simply copy itself to the sl array. This is because the rl[0] list is ordered. As a result, the sl array is equal to the _shift1 array.

Things change with the append command. The list used is `rl[1]` or `rho`. The value of variable `i` is 0. The first member of the `rho` list has a value of 7. The seventh member of the `_shift2` member is also 7, therefore the second list starts with a 7. In this case, another list is used to decide the contents of a new list.

One List within List Example

```
ch1811.py
pi = [(9*i + 5) & 15 for i in range(16)]
print pi
rr = [list(pi)]
print rr
```

Output
[5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12]
[[5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12]]

The program once again calculates a list of numbers from 0 to 15. The Bitwise Anding with 15 is to ensure that the resultant number is not larger than 15. A simple formula which is $9 * i + 5$ is used in place of random numbers.

Now this list is used to create another list, `rr`. This list has one member which is a list of 16 elements. The idea is to have a set of numbers that meet up to our rule of working with random numbers.

Finally Calculating Our Own RipeMD-160 Bit Hash

```
ch1812.py
import struct
import hashlib
rho = [7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]
pi = [(9*i + 5) & 15 for i in range(16)]

rl = [range(16)]
rl += [[rho[j] for j in rl[-1]]]
rl += [[rho[j] for j in rl[-1]]]
rl += [[rho[j] for j in rl[-1]]]
rl += [[rho[j] for j in rl[-1]]]

rr = [list(pi)]
rr += [[rho[j] for j in rr[-1]]]
rr += [[rho[j] for j in rr[-1]]]
rr += [[rho[j] for j in rr[-1]]]
rr += [[rho[j] for j in rr[-1]]]

f1 = lambda x, y, z: x ^ y ^ z
f2 = lambda x, y, z: (x & y) | (~x & z)
f3 = lambda x, y, z: (x | ~y) ^ z
f4 = lambda x, y, z: (x & z) | (y & ~z)
f5 = lambda x, y, z: x ^ (y | ~z)
fl = [f1, f2, f3, f4, f5]
fr = [f5, f4, f3, f2, f1]

_shift1 = [11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8]
```

333

```
h.update("A")
h = h.hexdigest()
if h == h1:
    print "All is Ok"
else:
    print "Vijay Mukhi is an imbecile"
```

Output

```
Hash is ddadef707ba62c166051b9e3cd0294c27515f2bc
All is Ok
```

In the world of the RipeMD, there are lists and lists and more lists. Why the algorithm does what it does is beyond our understanding.

This is the final program that calculates the RipeMD hash value. It may be a repetition of a lot of the code written earlier but let's start from scratch again. The task is to compute the hash value of message A. The actual hash is calculated in the RipeMD160 function. The message is manually created. Look at the last 8 bytes once again.

The same five constant variables are used starting with 0123. Then there are five chaining variables as the hash is 160 bits large. The variables a, b, c, d, e are not used individually but indirectly, packaged as a tuple. The five constants are stored in three tuples hl, hr and h. Then the list, w has a length of 16 words (512 bits) of the message, as before.

Now the rules change. In Sha1, there were 80 rounds used by the function and the constant was changed every 20 rounds. In RipeMD, the computation on the message is changed every 16 rounds though the rounds remain the same as in 80, as $16 * 5 = 80$.

The box function with six parameters is called twice. Since four of these six parameters use the loop variable called round, a different function or list is picked up during each iteration of the for loop. The lists KL and KR have a list of functions. When the value in the round variable changes, a different function gets called.

The hl variable or tuple will be the initial constant for the first time in the loop. Then it takes on the return value of the box function. In the box function, the chaining variables a, b, c, d and e get their values from the hl tuple.

The box function returns a tuple which is nothing but the values of the variables chaining variables a, b, c, d and e. In the box function, these temporary chaining values are initialized to hl or hr and then a different set of chaining variables are returned to be used again.

The fl list uses a different function for the same 16 rounds. This is because it has only 5 functions that can be called. In the main function ripemd160, the box function is called 5 times and in all there are 16 rounds, so $16 * 5 = 80$ rounds.

The KL list uses a different square root value for each round of 16 when compared to the KR list. The L and R start for left and right.

The variable w is the actual message, it gets its value from parameter s, which is the actual message. Not to be confused with the variable s, as a parameter in the box function. The rl and rr are different lists. To sum up, the data types given to the box function are tuple, function, int, tuple, list and list.

We accomplish a few things in the box function. A total of 16 rounds are executed on each set of data given to it. Then a value to T, a temp variable is computed which cannot exceed 32 bits. The chaining variable a is added to the value returned by function f. This function is given three chaining variables b, c, d. The function names are decided using the fl list, which contains the same Bitwise operations. The parameter k is added as well, which is obtained from the KL list, a list of the square root constants.

Next is to bring in the message word, but in an indirect manner. The `rl` list is used for an offset, though the `w` variable could be used directly. This would not make the output random enough. The `rl` list starts off as simple list but then ends up as a list of 5 list with 16 elements.

No rotations no hash. A simple rule. The rotation is used in the next step of calculating the variable `T`. Here, the list `s` has a different set of members each time and therefore the rotation is not the same for individual rounds. At the end of these rounds, we get one value for the left round, `hl` and another set of values for the second round in `hr`. The variable `h` is the final hash which is a tuple of 5 words. The result of $32 * 5$ is what gives a hash size of 160 bits.

The most complicated hash seen so far.

Another Stab at Explaining the RipeMD Hash

```
ch1813.py
import struct
import hashlib

rho = [7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]
pi = [5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12]
rl = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], [7, 4, 13, 1, 10, 6, 15, 3,
12, 0, 9, 5, 2, 14, 11, 8], [3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12], [1, 9,
11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2], [4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8,
11, 6, 15, 13]]
rr = [[5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12], [6, 11, 3, 7, 0, 13, 5, 10,
14, 15, 8, 12, 4, 9, 1, 2], [15, 5, 1, 3, 7, 14, 6, 9, 11, 8, 12, 2, 10, 0, 4, 13], [8, 6,
4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14], [12, 15, 10, 4, 1, 5, 8, 7, 6, 2, 13, 14,
0, 3, 9, 11]]
f1 = lambda x, y, z: x ^ y ^ z
f2 = lambda x, y, z: (x & y) | (~x & z)
f3 = lambda x, y, z: (x | ~y) ^ z
f4 = lambda x, y, z: (x & z) | (y & ~z)
f5 = lambda x, y, z: x ^ (y | ~z)
fl = [f1, f2, f3, f4, f5]
fr = [f5, f4, f3, f2, f1]
sl = [[11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8], [7, 6, 8, 13, 11, 9, 7, 15,
7, 12, 15, 9, 11, 7, 13, 12], [11, 13, 6, 7, 14, 9, 13, 15, 14, 8, 13, 6, 5, 12, 7, 5],
[11, 12, 14, 15, 14, 15, 9, 8, 9, 14, 5, 6, 8, 6, 5, 12], [9, 15, 5, 11, 6, 8, 13, 12, 5,
12, 13, 14, 11, 8, 5, 6]]
sr = [[8, 9, 9, 11, 13, 15, 15, 5, 7, 7, 8, 11, 14, 14, 12, 6], [9, 13, 15, 7, 12, 8, 9,
11, 7, 7, 12, 7, 6, 15, 13, 11], [9, 7, 15, 11, 8, 6, 6, 14, 12, 13, 5, 14, 13, 13, 7, 5],
[15, 5, 8, 11, 14, 14, 6, 14, 6, 9, 12, 9, 12, 5, 15, 8], [8, 5, 12, 9, 12, 5, 14, 6, 8,
13, 6, 5, 15, 13, 11, 11]]
KL = [0, 0x5a827999, 0x6ed9eba1, 0x8f1bbcdc, 0xa953fd4e]
KR = [0x50a28be6, 0x5c4dd124, 0x6d703ef3, 0x7a6d76e9, 0x0]
def rol(s, n):
    return ((n << s) | (n >> (32-s))) & 0xffffffffL
def box(h, f, k, w, r, s):
    (a, b, c, d, e) = h
    for word in range(16):
```

```
Output
Hash is ddadef707ba62c166051b9e3cd0294c27515f2bc
All is Ok
```

These four initializations are to be performed only once for every 16 iterations. However, the fl or f variable of the past needs to be computed every time with a new round. The function may change every 16 rounds but the value is calculated afresh as the variable b , c and d change.

The right round is more readable. See the change in size. The chaining variables get initialized once every 16 rounds like before, but handled separately.

One look at the code, it follows the same structure as the earlier hashes. It can be concluded that the RipeMD code has more randomness than the Sha family. Unlike the Sha hash, a list is used to give values. In the program, we tried to simplify the RipeMD hash code as much as we could.

Now let's move on the world of hashes in general.

One question that is not yet answered is that why rotate variable a by 5 and variable b by 30 or right rotate by 2.

These are very small values. One group did research which said that these were the smallest values of rotation that would preserve the security of the hash. This is also because the Sha-1 hash uses 80 rounds, which is big. If the number of rounds was lower, then the rotations would be larger. The paper talks of calculating hamming weights, something beyond us. A hamming weight represents the number of 1 bits in a number. In Python, one way of calculating a hamming weight of the number 9 would be `bin(9).count("1")`.

There is a concept of hamming distance also, but the results could not be verified by us, so it's been ignored. The paper says higher the hamming weight better the security and lower the probability of finding collisions. We tried but failed to understand it.

The key take away is that if 80 rounds are performed, then the rotations or the constants do not matter much. Also, what we have learned so far is that the number of rotations do matter, as in Sha-0. So, do the number of rounds.

In the good old days, constant time shifters or rotators were not implemented in hardware or microcode. These had chips like the Intel i286 or the Pentium IV. It would take a year to calculate the hash. In the earlier days, the more the bits were rotated, the slower the process.

These days, chips are lighting fast so there is nothing to worry about rotation speed. Hashes designed for today follow a different rule but the flip side is that these hash algorithms must now run in your fridge also. A newer and more serious problem. Fridges get cold, hash calculations become slow. A modern-day cryptography problem. Also, when a fridge defrosts, hashing also stops. This is what hashes do to our intelligence.

The greater the variations and number of rotations makes life more difficult for the attacker.

Security Issues

There are only two types of security issues with hashes.

The first is called a pre-image attack. In this type of attack, a hash value is available and the task is to find the original message that created it. As hashing is one way street, it is next to impossible to find the message that created the hash value.

The second attack is a collision attack. Can we find any two messages that generate the same hash value? Here, what is important is the collision and not the actual message.

There is no known methods for a pre-image attack, but we can surely narrow down the number of attacks for a collision attack. If finding a collision is 2^{64} operations, then for a pre-image attack, it would be 2^{128} operations. Picked this up from some page when understanding hashes.

A 160-bit number is a number between 0 and 1.46×10^{48} .

Cryptographers love talking about the Birthday Paradox. The logic is as follows. I was born on the 7th of December 1957. Everyone on Facebook also knows this. I would have to ask about 253 people before I came across someone who was born on the same day 7th December.

Yet, I would only have a 50% chance of finding someone who was born on the day I was born.

However, in a group of 23 people, the chances of 2 people in this group sharing the same birthday is 50%.

This is the difference between the collision attack and the pre-image attack, which is many times slower, as the objective is to find a certain specific message.

Applying the birthday paradox to the 160 bit Sha-1 hash, then about 2 raised to 80 evaluations would be required to find a collision. Cryptographers live on a different plateau. So, if anyone finds a way to reduce the number of evaluations to below 2 raised to 80, the hash is said to be broken.

On these grounds, Sha-1 is broken even though no one has publicly disclosed a real collision. So, when someone writes about a hash been broken, ask for two messages that give the same hash.

Changing a single bit of a message changes the message digest or hash drastically. Mathematicians would say that the message digest would change with a high probability. So, there is always that next to impossible chance that find a collision.

The Message

The reason behind adding a 1 at the end of the message is to avoid collision. A message with lots of 0's at the end would lead to a collision.

In all the hashes seen so far, the output of the first round is used as the input for the second round. This method of calculating the message digits is called a domain extender or the Merkle-Damgard scheme. Merkle is one name one can never forget.

The main weakness of the hashes seen so far, except for the RipeMD-160 hash, is that a large amount of data is generated, from sixteen 32 bits to eighty 32 bits, it is an increase of 2048 bits. The end goal for extending the bits is to ensure that the regularities in input data can be randomized. By the way, this process of increasing bits is called recurrence and bit rotation. It results in a bigger search space.

If there is no rotate left on these extra bits, then the words with any of the extra bits depend only upon the previous bits.

Attackers find collisions by using differential paths. Just rotating a bit by 1, makes this concept, differential paths harder to use. When the hash goes through 80 or 64 rounds, it is called a state update transformation. For whatever it may count, Sha-1 is 30% heavier or slower than MD5 and Sha-256 is twice heavier than Sha-1, especially on a GPU.

These lines are randomly picked up from various Internet sites while understanding the hash algorithms. We assume they are true as we have no code to back us up. There is no code to confirm our narrative so do not get into a slanging match with us, once again we are not experts at mathematics.

CHAPTER 19

ECC with Sage - Part 1

A simple confession. This chapter is again one of the most difficult chapters we have ever written, not in this book but in our entire lives. Ask anyone in the world and they will tell you that Elliptic Curve Cryptography (ECC) is the most difficult part of cryptography. They will also add that cryptography is the most difficult subject to understand in the solar system. Bitcoin uses ECC to sign transactions.

Using Python code by itself would make it very difficult to explain ECC. So, we use an open source Math or package called Sage. Sage can be downloaded off the internet and even though it is interactive like Python, we will write our code in Python and execute it using the Sage package. It is 100 % free and runs everywhere. Sage has an extremely verbose documentation, and it is written by Mathematicians for Mathematicians.

When we downloaded SageMath, it creates a Volume starting with sage-. Within this volume, we have a SageMath folder, which has an executable called sage. We have moved the entire folder SageMath to the Desktop, but you can decide on your location. On the Mac, the PATH variable is set to include this folder, so we can execute it from anywhere. Sage simplifies using ECC and large numbers computations. It has nothing to do with Bitcoins.

The first time when sage is executed, it does some weird things and then it will be very slow, so be patient.

Let's start with a very simply program in Python that uses Sage.

Simple Check to See that Sage is Working

```
ch1901.py
from Sage.all import *
print "Hell"
```

Output
Hell

The program is executed using the following command:

```
$sage -python ch1901.py
```

The program imports a python library that acts as a bridge between Python and Sage. Sage adds its own functions which integrate fully with Python code. The python print command is executed and hence the output. It's not for us to figure out who executes it, whether Python or Sage.

Python cannot run the above program as it cannot locate the module sage.all which is imported using the from keyword, it will throw an error. So, the sage command is used. There is a wait for a few seconds when the program is executed.

Sage and Finite Fields

```
ch1902.py
from sage.all import *
P = FiniteField(3)
print P
for i in range(0,12):
    print P(i) ,
print "\nCardinality of P is %d" % P.cardinality()
```

Output

```
Finite Field of size 3
0 1 2 0 1 2 0 1 2 0 1 2
Cardinality of P is 3
```

The variable P stores the value returned from the FiniteField function. This function is the first function we are using from the Sage toolkit. The good old print function is used to display its value. As per the output, P is a Finite Field of size 3. The answer is 3 because 3 was given as a parameter to the FiniteField function.

A finite field is simply a set of consecutive numbers. These numbers have an upper limit or a target value, in our case 3. The variable P represents the values 0, 1 and 2. The value 3 itself, is not part of this set of numbers.

The for statement is used to print the values represented by P. To our surprise, the for loop iterates 12 times, starting from 0 to 11. Subsequently, the individual values of a finite field are displayed using round function brackets. The loop variable i has a value starting from 0 and ending at 11.

For the initial values of the variable i, 0, 1 and 2, produce no surprises. The values returned are 0, 1 and 2. However, an error message is expected when the index value is 3 as there is no member 3, but the values rotates around and starts from 0 again. This is the whole concept of modulo arithmetic.

In this case, our FiniteField P has length of infinity but it's values can only be 0, 1 and 2; one less than the prime number passed to the function.

Finite Fields work with prime numbers. In other words, a prime number has no factors at all. Pass a non-prime number like 2 or 4 and you will see something very different.

The Mathematicians use the cardinality word everywhere. It simply refers to a count value, as in how many members are there. So, if three friends are out for a movie, the cardinality is 3. Nearly everything that Sage creates, has a cardinality function that gives a count of the members in the object. A more glorified len if you may.

Elliptic Curves

An Elliptic Curve however has nothing to do with Ellipses. To put it very simply, an Elliptic curve can be represented by a simple equation.

$$y^2 = x^3 + ax + b$$

Changing the value of x will give a value of y and an Elliptical curve can be created. This simple equation is what the Elliptical Curve Cryptography is based on. There are some zillion Elliptic curves that can be used because the values of a and b can be different. The Bitcoin world uses a very effective Elliptic Curve called secp256k1. In this Elliptic Curve, the value of a is 0 and the value of b is 7. The above curve now becomes

$$y^2 = x^3 + 7$$

The next program uses Sage to create an Elliptic curve.

A secp256k1 Elliptic Curve

```
ch1903.py
from Sage.all import *
P = FiniteField(11)
E = EllipticCurve(P, [0,7])
print E
```

Output

Elliptic Curve defined by $y^2 = x^3 + 7$ over Finite Field of size 11

The EllipticCurve function creates an Elliptic Curve. One of the parameters to the function is a FiniteField object, which defines a set of consecutive numbers. Then, this function requires a list with the values of the constants a and b , which for the secp256k1 curve is 0 and 7, in that order.

The value of E when displayed shows that the Elliptic curve is denoted by the equation $y^2 = x^3 + 7$ and the Finite Field is 11. The number 11 is for good luck purposes, you can substitute it with your lucky prime number. We all have lucky prime numbers.

The Cardinality of a secp256k1 Elliptic Curve

```
ch1904.py
from Sage.all import *
prime = 11
P = FiniteField(prime)
E = EllipticCurve(P, [0, 7])
print E.points()
k = 0
print
for x in range(0, 11):
    for y in range(0, 11):
        lhs = (y * y) % (prime)
        rhs = ((x * x * x) + 7) % (prime)
        #print "x=%02d y=%02d lhs=%02d rhs=%02d %s" % (x, y, lhs, rhs, lhs == rhs)
        if lhs == rhs:
            print("(%d : %d)" % (x, y),
            k = k + 1
print
print "Cardinality EEC is %d" % E.cardinality()
print "Cardinality Ours is %d" % k
```

Output

```
[(0 : 1 : 0), (2 : 2 : 1), (2 : 9 : 1), (3 : 1 : 1), (3 : 10 : 1), (4 : 4 : 1), (4 : 7 : 1), (5 : 0 : 1),
(6 : 5 : 1), (6 : 6 : 1), (7 : 3 : 1), (7 : 8 : 1)]
(2 : 2) (2 : 9) (3 : 1) (3 : 10) (4 : 4) (4 : 7) (5 : 0) (6 : 5) (6 : 6) (7 : 3) (7 : 8)
Cardinality EEC is 12
Cardinality Ours is 11
```

The Elliptic Curve Object E has a function called points that gives us all the points that fall on this Elliptic Curve. There are 12 points on this Elliptic Curve, based on the prime field of 11. The cardinality function on the second last line agrees with this value.

Let's check if any point (x, y) lies on this curve. So, we take all possible combinations of x and y that range from 0 to 10. The prime field ends at 11, the chosen maximum value of the prime number.

At first, the left-hand side is calculated by squaring y. The mistake most people make is that if the value of y is 5, the square is 25. This is incorrect. The next step is to mod the y squared value with the prime number 11 as the value cannot be larger than 10. This is an inherent concept of a prime field, there is an upper limit.

In the same fashion, the right-hand side is calculated by first cubing x and then adding 7. It is a compulsory to mod everything by 11 or the chosen prime number. The values of x and y are on the Elliptic curve. So, a combination of x and y is searched where both sides match. Every time a match is found, the variable k is incremented by 1. The value is then displayed. Only two coordinates are displayed even when the Elliptic Curve has 3 coordinates, x, y and z.

In all, there are a total of 11 points where the left-hand side equals the right-hand side. This is one less than what the Sage computes. The reason being that the point at infinity which is a point on every Elliptic Curve has not been considered. When the value of z is 0, that point (0:1:0) is the point at infinity. There is only value when the z coordinate is 0.

It takes an infinite amount of time to understand a point at infinity so we will skip it for the moment.

Creating a Point on the Elliptic Curve

```
ch1905.py
from Sage.all import *
prime = 11
P = FiniteField(prime)
E = EllipticCurve(P, [0, 7])
G = E.point((2,9))
print G
G = E.point((2,8))
print G
```

Output

Traceback (most recent call last):

(2 : 9 : 1)

Coordinates [2, 8, 1] do not define a point on Elliptic Curve defined by $y^2 = x^3 + 7$ over Finite Field of size 11

The earlier program unraveled the fact that the point 2,9 lies on the Elliptic Curve. The Elliptic Curve Object E has a function called Point, which given the x and y coordinate as a tuple, returns an Elliptic Point object.

When this object G is displayed, it shows the values of x and y and z. These are (2:9:1). So far so good. Next is to create an Elliptic Curve point object by using x and y coordinates of 2 and 8, knowing that they do not fall on the Elliptic curve. Sage comes back with a runtime error or exception saying that the point (2,8,1) is not a valid point on the Elliptic curve.

All these steps are taken to basically discern how Bitcoin takes a private key and gives a public key, which is a point on the Elliptic curve. The next phase it to learn how Bitcoin uses EEC to sign a transaction. It verifies whether a certain private key signed a transaction without having any access to that private key.

ECC Multiplication is Not Multiplication but Still a Multiplication

```

ch1906.py
from Sage.all import *
prime = 11
P = FiniteField(prime)
E = EllipticCurve(P , [0 , 7])
G = E.point((3 , 1 ))
print G
G1 = G * 2
print G1
G = E.point((2 , 2 ))
print G
G1 = G * 2
print G1
print G
G = E.point((2,2))
G1 = G * 12
print G1
G = E.point((2,2))
G1 = G * 13
print G1

```

Output

```

(3 : 1 : 1)
(3 : 10 : 1)
(2 : 2 : 1)
(5 : 0 : 1)
(2 : 2 : 1)
(0 : 1 : 0)
(2 : 2 : 1)

```

The point object G is a x and y value. It is also a point on the Elliptic Curve. Let's take the point 3,1 and multiply it by 2. Conventional logic would say that the resultant point will be 6,2. The problem is that this point 6,2 would normally not be on the Elliptic Curve at all. So, the concept of multiplication does not make any sense in the Elliptic Curve universe.

An Elliptic Curve point object can be multiplied by any number, here G, which represents the point 3,1 is multiplied by 2. Surprisingly, the output is 3,10 which by the way is a valid point on the EC. When another point 2,2 is multiplied it by 2, a new point on the curve is obtained but this time the point is 5,0.

So, we can use the multiplication sign to multiply a point on the Elliptic Curve, but the resultant value is beyond our understanding.

The cardinality or number of points on the curve is 12. So, if a point 2,2 is multiplied by 12 it will give a point at infinity, which is 0,1,0. However, multiplying the same point 2,2 by 13 which is one number higher than the prime number used in creating the prime field, it eventually comes to the same point, 2,2.

This makes sense as the number of points is decided by the number given to FiniteField function. So, when the multiplication is with a larger number, it will run out of valid points on the curve and land up at the same point again and again. The cardinality decides at which point it returns to square one or when the chickens come home to roost.

Thus, EC does support multiplication, but not the way we understand it. It has nothing to do with multiplication except for the * symbol. This is where the confusion comes in.

The secp256k1 Elliptic Curve has some Zillion Points

```
ch1907.py
from Sage.all import *
prime = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
print "%x" % prime
print "%d" % prime
P = FiniteField(prime)
E = EllipticCurve(P, [0, 7])
print E
print E.cardinality()
```

```
Output
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffeffffc2f
11579208923731619542357098500868790785326998466564056403945758400790883
4671663
Elliptic Curve defined by  $y^2 = x^3 + 7$  over Finite Field of size
11579208923731619542357098500868790785326998466564056403945758400790883
4671663
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
```

The Elliptic Curve used by Bitcoin follows the standard secp256k1 curve, which uses very large prime numbers; up to now the programs had small numbers. This curve named secp256k1 was not designed by the Mr. Satoshi. This is a NIST standard mandated by the Government of the United States of America. Mr. Satoshi simply liked this curve more than others and hence it is the chosen one. Yes, some people have a favorite Elliptic Curve. Choosing these numbers make it more secure than the other Elliptic Curves.

The prime number is calculated by using powers of 2. Displaying them as powers of 2 shows us visually an answer we can relate to. This number is a very random number. The large number has no special significance. Therefore, it is called a random number, or Nothing Up My Sleeve number.

Anytime a constant value is used in cryptography, the first question asked is why have such constants been chosen? Have they created a backdoor so that the cryptography can be broken?

The cardinality of our Elliptic curve now gives more than some zillion points. The greater the number of points on the curve the more difficult it becomes coming to the same point again, especially when a point is multiplied by a number.

The cardinality of a curve is not the same as the prime number that created the curve.

Selecting a Random Point on an Elliptic Curve

```
ch1908.py
from Sage.all import *
prime = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
print "%x" % prime
print "%d" % prime
P = FiniteField(prime)
```



```

E = EllipticCurve(P , [0 , 7])
print E
Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
G = E.point((Gx , Gy))
print "%x" % G[0]
print "%x" % G[1]

```

Output

```

fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffc2f
11579208923731619542357098500868790785326998466564056403945758400790883
4671663
Elliptic Curve defined by  $y^2 = x^3 + 7$  over Finite Field of size
11579208923731619542357098500868790785326998466564056403945758400790883
4671663
79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8

```

Two random values for x and y are used. These values are very special numbers, which will be explained later.

A point on Elliptic Curve is created using these two variables Gx and Gy. Surprisingly, no error is reported, which only means that Gx and Gy is a valid point on our Elliptic Curve.

To reconfirm, the x and y of the G point are printed and their values match the ones in variables Gx and Gy.

A Public Key is Nothing But a Point on an Elliptic Curve

```

ch1909.py
from Sage.all import *
prime = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
P = FiniteField(prime)
E = EllipticCurve(P , [0 , 7])
Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
G = E.point((Gx , Gy))
d = 0x4200000000000000000000000000000000000000000000000000000000000000
pd = G * d
print "%x:%x" % (pd[0] , pd[1])

```

Output

```

f9a7699db2eec6a4116373fde8e5c7f46af3d81c96b912cdc0e2dd67d00ed6d7:81888bc
7c39617d4f7c3437b4ace461402174ed76561090838123f952d41488

```

We will explain this program along with the next program. This program has to be executed the old way using python.

```

ch1910.py
import pybitcointools
privkey = '4200000000000000000000000000000000000000000000000000000000000000'
pubkey = pybitcointools.privtopub(privkey)

```

```
print pubkey
```

The public key of a private key starting with 420 using the pybitcointools library

Output

```
04f9a7699db2eec6a4116373fde8e5c7f46af3d81c96b912cdc0e2dd67d00ed6d7081888
bc7c39617d4f7c3437b4ace461402174ed76561090838123f952d41488
```

The program uses a very large prime number as the FiniteField. There is a reason behind using only this number. Also, two very large numbers Gx and Gy are used for a point on the Elliptic Curve.

Then a variable d is used to store the private key, beginning with 420. Next, the Point G is multiplied with the private key stored in variable d. Again, this is not a Multiplication but ECC Multiplication.

This multiplication returns a tuple which has two values, the x and y coordinate. These numbers when displayed, look very familiar. To jog our memory, it is like the program that calculated a public key. The public key has a value similar to the one generated by a private key, starting with 420. Surprisingly, the public key created by the pybitcointools library is nearly identical to the public key calculated using Sage.

The first byte, 04 is removed from the newly generated public key. Also, a 0 is added to the y co-ordinate to give it the right length.

This could only mean that the public key is simply a point on our Elliptic Curve. The point G is called a generator point and its Gx and Gy values are random numbers. They are constant only for the secp256k1 curve.

Once again, the Generator point is multiplied with the private key to get another point on the Elliptic curve which is the public key. It is very easy to generate this public key. Everyone in the solar system knows of this Generator point, it is public knowledge.

We read a million times on the web that a public key is a point on the Elliptic Curve, but only when we calculate the public key ourselves does this concept sink in. Doing is believing.

The 04 prefix of the public key is explained later.

Hardcoding the Number of Points on the secp256k1 Elliptic Curve

```
ch19011.py
from Sage.all import *
prime = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
P = FiniteField(prime)
E = EllipticCurve(P, [0, 7])
N = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
print N
print E.cardinality()
print E.order()
```

Output

```
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
```

There are many constants used in our Elliptic Curve journey. This curve is called the secp256k1, because of the constants chosen. Another curve will have its different chosen constants.

In our code, the variable N represents the number of points on the Elliptic Curve. Once the constant, a and b and the FiniteField prime are determined, then the number of points on the curve are fixed. The cardinality function or order function returns this number. Most programs do not use these functions but simply initialize variable N to this large value. Once again, N is the number of points on the Elliptic Curve and it is a very large number and it is equal to the cardinality and order.

Random Numbers with Sage

```
ch1912.py
from Sage.all import *
prime = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
P = FiniteField(prime)
E = EllipticCurve(P, [0, 7])
N = FiniteField(E.cardinality())
print N.random_element()
print E.random_element()
print N(20)
print type(N(20))
```

Output

```
79824689525906975811501836002967846317697419953704677717266062361309442
149136
(22240768267882517235144697906758637209437148267972122673872308352141338
0374 :
61268332541411179230621152792648336858032067849158375993951024479214828
268470 : 1)
20
<type 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>
```

One very useful function offered by Sage is called `random_element`. It can generate a random number on a finite field or on a curve. It is part of the functions offered by a `FiniteField` type object. However, an `Elliptic Curve` also has a function with the same name, `random_element`. The difference is that `FiniteField` returns a random number whereas the `Elliptic Curve` function returns a random point on the `Elliptic Curve` itself. It is pertinent to note that lack of a good quality random number generator will break ECC. We have burnt our Bitcoins because we do not practice what we preach.

Working with ECC requires ECC objects and not Python objects. A number 20 given to the `FiniteField` variable N will not return 20 as a long type, but a Sage object. These Sage object types names are very easy to remember.

Modulo Inverse

```
ch1913.py
prime = 7
for i in range(prime):
    for j in range(prime):
        if (i * j) % prime == 1:
            print "Mod Inverse of %d is %d (i*j) %d" % (i, j, (i*j) )
```

Output

```
Mod Inverse of 1 is 1 (i*j) 1
Mod Inverse of 2 is 4 (i*j) 8
Mod Inverse of 3 is 5 (i*j) 15
Mod Inverse of 4 is 2 (i*j) 8
Mod Inverse of 5 is 3 (i*j) 15
Mod Inverse of 6 is 6 (i*j) 36
```

Lets start by openly admitting that we are not mathematicians nor have any plans to be one. There is a concept called mod inverse in mathematics, it is used heavily in cryptography. The mod inverse is calculated over a prime field and a prime field is defined by a chosen prime number. We use a prime number like 7 because it is very small number.

A mod inverse of a number (A) is determined when a mod operation of the number(A) multiplied by another number(B) results in a 1. This other number(B) is called the mod inverse.

To find out the mod inverse of 2, 2 is multiplied by 4 to get 8. Mod this value by 7 and the result is 1. Thus, the mod inverse of 2 is 4. Here the values are (mod)ed with the prime number 13 instead. Change the prime number and the mod inverse obviously changes.

In our code, every number in the outer for loop is multiplied with every possible number in the inner for loop. Since, we are dealing with the concept of Finite Fields here, after the multiplication, a mod (%) is performed. The numbers cannot be larger than the prime number used to create the Finite Field.

This is one very slow way of getting the mod inverse of a number over a finite field. Therefore, Sage comes in.

Mod Inverse using Sage

```
ch1914.py
from sage.all import *
n = FiniteField(7)
w = 1 / n(2)
print w
w = 1 / n(5)
print w
```

Output

```
4
3
```

The value of n(2) would be the third element of the FiniteField n. Now in modulo arithmetic, there is no concept of a number with decimal places, only integers or whole numbers. Hence, the use of division in modulo arithmetic is disregarded. The only way out in the above case is to find the modulo inverse.

When Sage sees that a number is divided into a finite field, it uses code like the one given above. Sage is smart and returns the modulo inverse. It bears in mind that in a Finite Field, the number cannot be larger than the prime number.

The answer with Sage and our code, both are right as the mod inverse of 2 is 4 for a prime number 7. The mod inverse of 5 is 3. This is incorrect unless the finite field is pre-stated.

A little later, we will delve into some more mathematics of the modulo inverse but for now the focus is exclusively on the processes used by Bitcoin to sign and verify a message.

Prior to that let's understand these processes without using any code. Then, we will implement the actual Python Sage code to replicate the same.

The following variable names are used in our Python code.

dA: This is our private key. It will be a small number like 3 so that the code can be deciphered using a calculator in your head. This value was 420 earlier.

G : This is not a random point on the Elliptic Curve but an especially chosen point for the secp256k1 curve. In our case, just kidding.

QA : This is the public key which is a point on the Elliptic Curve. The point value is calculated by multiplying the Generator point with the private key.

m: The actual message to be signed. In real life, it would be the Bitcoin Transaction bytes. It is not used directly in our code.

e: This is the Sha256 hash value of the message. In our case, it is a small number and not a 256-bit number.

n: This is the order or cardinality or the number of points on our Elliptic Curve. It is fixed as the Elliptic Curve Bitcoin uses it.

k: The ECC rests on only one shoulder, a good quality random number generator that must not be reused.

The value in this variable changes each time a signature is created. Therefore, the signature of the same Bitcoin transaction will always be different.

R: When there is an ECC multiply of our random number k with the Generator Point G, another point on the Elliptic Curve is acquired, that point is called R.

r : The EC signature is two distinct numbers, the first is called r.

s : It is the second number, the signature. These two values r and s are part of the input script value.

These variables are used to verify the signature.

There are 4 values: the hash e, the r and s that the signature generating process creates and the public key of the signer. The variable w stores the mod inverse value of s. The variables, w, r and s are made public to the whole world.

u1: A temporary variable to store the multiplication of the hash e, which is also public like w. It is multiplied with G.

u2: Another temp variable which stores the multiplication of the r value generated during signing and the inverse modulus of s or the variable w. It is multiplied by QA.

P: A Point that is an indicator to the final answer. The temp variables u1 and u2 are added.

Finally, the r1 value is the x value of the point P which will be the same as the r value of the signing process. If they are the same, then the person who signed the data with their private key is the same person who generated the public key from this private one.

Now let's look at the above variables in action.

```
e = SHA256HASH(m)
```

The step of converting the original message m into a hash value e is skipped.

```
0 < k < n
```

Also, it is ensured that the random number is greater than 0 and less than the number of points on the Elliptic curve n. Sage generates a random number within the specified range.

```
R = (rx , ry) = k * G
r = rx mod n
r > 0
```

The Generator Point G is ECC multiplied by an integer k (a random number) to give another point R. A point is normally x, y and z coordinate. The z axis value is removed. The value of z is checked, if it is 0 then we abort.

The value of r is only the x coordinate. This is one more point on the Elliptic Curve that depends upon the random number. The value of r depends upon the random number and the Generator Point.

```
s = (e + dA * r / k) mod n
s = ((e + dA * r) * inversemod(k)) mod n
```

The next task is to generate the actual signature s. So far, the message hash e is not used at all. So, first the private key is multiplied with the value of r, recently calculated. The message is added to this value.

As there is no division in ECC, the random number is multiplied with the mod inverse of k. This mod inverse is handled by Sage internally.

To sum up, the value of r is mainly dependent upon the random number k. The value of s is dependent on the data e and the private key dA. Therefore, the Bitcoin signature varies each time it is calculated with the same data.

The values of r and s are made public as part of the input script. The hash e and the public key is always public. But the private key is not made public.

Now for the actual verification.

```
w = 1 / s
w = inversemod(s)
```

First task is to find the inverse modulus of s which Sage handles internally.

```
u1 = (e * w) * G
u2 = (r * w) * QA
```

Then a value of a temp variable u1 is calculated where the message is multiplied with w, the inverse mod. The second temp variable u2 is calculated which is again r, the second value from generating the signature again with w.

Now a point P which is the ECC addition of two points is calculated. The values in u1 and u2 are added. The final answer is simply one more point on the Elliptic Curve.

The final value of r is the x coordinate of the point P. This must be equal to the random value returned by the signing process. The signature thus, is verified without having laid our hands on the private key.

Our Python code explains this concept better.

Creating and Verifying the ECC Signature

```
ch1915.py
from sage.all import *
prime = 13
P = FiniteField(prime)
E = EllipticCurve(P , [0 , 7])
print "Points on the Elliptic Curve are " , E.points()
n = FiniteField(E.cardinality())
```

```

print "Cardinality or Number of points on the Elliptic Curve " , E.cardinality()
print "n is not a simple number but a " , n
G = E.point((7 , 5))
print "Original Generator point G on the Elliptic Curve as chosen by us " , G
dA = 3
print "Our Private Key d is " , dA
QA = G * dA
print "Our generated Public Key or Point on the Elliptic Curve after EC Multiply is " , QA
e = 4
print "Hash of message e is " , e
print "=====Signing Starts"
k = n.random_element()
if k == 0:
    print "As the random number is 0 e cannot calculate the signature s. Aborting..."
    exit(0)
print "Current random number k generated over the Finite Field is " , k
R = int(k) * G
print "After EC Multiply Generator Point with Random Number. New Point on Elliptic Curve " , R
(x , y , z ) = R
if z == 0:
    print "Point at Infinity Aborting..."
    exit(0)
r1 = R.xy()
print "After removing the z coordinate from the New Point " , r1
r = r1[0]
print "After extracting the x value from the New Point the final r value " , r
print "The random number from the Finite Field using r as the index n(r) " , n(r)
modinversek = 1 / k
print "The mod inverse of k is " , modinversek
s = ( 1 / k ) * ( e + n ( r ) * dA)
print "The signature s is " , s
print "The index in n using s n(s)= " , n(s)
if n(s) == 0:
    print "Aborting as n(s) is 0..."
    exit(0)
print "-----The final Signature that we compute s " , s , " and r is " , r
print "=====Verifications Starts"
w = 1 / n(s)
print "The mod inverse w using the signature s is " , w
print "w is not a number but a " , type(w)
u1 = int ( w * e ) * G
print "Multiplication of w and e " , w * e
print "First Temp calculation Point on the EC is " , u1
u2 = int ( n ( r ) * w ) * QA
print "Multiplication of n(r) and w " , n(r) * w
print "Second Temp calculation Point on the EC is " , u2

```

```
P = ( u1 + u2 )
print "Verification answer or adding the above two points is " , P
r1 = P.xy()[0]
print "-----The value of r after Verification r1 " , r1
print "The two r's are the same " , r == r1
```

Output

```
Points on the Elliptic Curve are [(0 : 1 : 0), (7 : 5 : 1), (7 : 8 : 1), (8 : 5 : 1), (8 : 8 : 1),
(11 : 5 : 1), (11 : 8 : 1)]
Cardinality or Number of points on the Elliptic Curve 7
n is not a simple number but a Finite Field of size 7
Original Generator Point G on the Elliptic Curve as chosen by us (7 : 5 : 1)
Our Private Key d is 3
Our generated Public Key or Point on the Elliptic Curve after EC Multiply is (11 : 8 : 1)
Hash of message e is 4
=====Signing Starts
Current random number k generated over the Finite Field is 6
After EC Multiply Generator Point with Random Number. New Point on Elliptic Curve (7 : 8 : 1)
After removing the z coordinate from the New Point (7, 8)
After extracting the x value from the New Point the final r value 7
The random number from the Finite Field using r as the index n(r) 0
The mod inverse of k is 6
The signature s is 3
The index in n using s n(s)= 3
-----The Final Signature that we compute s 3 and r is 7
=====Verifications Starts
The mod inverse w using the signature s is 5
w is not a number but a <type 'sage.rings.finite_rings.integer_mod.IntegerMod_int'>
Multiplication of w and e 6
First Temp Calculation Point on the EC is (7 : 8 : 1)
Multiplication of n(r) and w 0
Second Temp Calculation Point on the EC is (0 : 1 : 0)
Verification answer or adding the above two points is (7 : 8 : 1)
-----The value of r after Verification r1 7
The two r's are the same True
```

This program signs a hash value or some data that could be a Bitcoin transaction. The process of signing and verification outlined above is used.

Let's understand the code one line at a time.

The variable prime stores a very small prime number 13. The next program takes the prime number used by the Bitcoin secp256k1. The Elliptic Curve called E is created with only 7 points on this curve. They have been displayed on the screen. All the points computed in future must be one of these 7 points.

The variable n returned by the Finite Field decides the number of points on the Elliptic Curve. By using variable n, there is no need to worry about the range of values a random number can take or its modular inverse. It must be noted that n is not a normal number, it is special to Sage.

Our Generator Point G is a very small value whose x coordinate is 7 and y coordinate is 5. This Generator Point G is the next point on the Elliptic Curve after the point at infinity.

The private key variable dA has a value of 3. To acquire a public key QA, we ECC multiply the private key dA and Generator Point G. The resultant Point is (11, 8) which is the last point displayed on the Elliptic Curve. It is only because an Elliptic Curve is built with so few values, it is possible to see all the actual points present on the Elliptic Curve.

The message hash e has a value of 4. It's time to start signing this message hash.

First, a random number k from the FiniteField n is created. The function `random_element` is used for this purpose. The advantage here is that the random number value will never exceed the prime number of the Finite Field. A bonus, the modular inverse is taken care off.

The first error check is on the value of k, it must not be 0. It is Sage's responsibility to ensure that the higher value does not exceed the cardinality of the Elliptic Curve. This program generated a random value of 6, your mileage will vary.

The variable k is not an integer, so it is type-casted to an integer. Then, the ECC multiplication gives another point on the Elliptic Curve. This point R has a value of (11, 8). Once again and we will not repeat this, your mileage will vary. We confirm that R is a valid point on the Elliptic Curve.

Any point object has three coordinates, x, y and z. The z coordinate is mainly to check for the point at infinity. If z has a value of 0, then R is the point at infinity. If so, then the program must exit.

The tuple notation is used to extract the x, y and z coordinates. The `xy` function of a point object gives the x and y coordinates and it is saved in variable r1. The x coordinate is the value of the r variable that is shared with the world. The `[]` notation is used to extract its value. The current value of r is 7 which is the x value of the point on the Elliptic Curve.

The number 4 or `n(4)` is used to reference a finite field which give the same answer. They are however of different data types.

It is a little tedious to obtain the value of s. First, the mod inverse of k is computed. This value is 6 and it is stored in the variable `modinversek`. This magic is done by Sage. Then the private key is multiplied with the value of r (to be fair `n(r)`) recently calculated. The message hash is added to the above multiplication. Finally, this result is multiplied with the mod inverse of k. The signature s has a value of 3.

The two values required for verification are variable r and s, as in 7 and 3. This is the actual ECC signature for kids. The value of `n(s)` must not be 0, a check is performed here.

The next step is to verify if the private key dA signed the document. The first requirement is the mod inverse of `n(s)` in a variable called w. The value of s is 3 but its mod inverse is 5. Like variable n, variable w is also special. The first tmp variable u1 is a Point on the Elliptic Curve. So, first the Generator Point G is ECC multiplied with result of the multiplication of the mod inverse of s i.e. w with the message hash e. This new point u1 is (7,8). Close to the original Generator point, this cannot happen with large real-life values.

As there are very few points on our Elliptic Curve, we will be reusing some of them. The number obtained from the result of multiplying w and e is casted into an integer for ECC multiplication. Another point, u2 is calculated on the curve by ECC. This is achieved by multiplying the public key with the result of the mod inverse of s i.e. w and the value of `n(r)` (n was calculated in the signing stage). The resultant point on the curve is (0,1).

These two points, u1 and u2 are added on the Elliptic Curve. This is like a ECC multiplication, it is not a real addition. The final point P (7,8) is another point on the Elliptic curve.

The xy values are extracted using the function `xy` and then the x-value is retrieved using the tuple notation as before. The final x value and the value of r must be the same, 7. If they are not the same then it is assumed that the public key used was not generated by the private key that initially signed the message.

The program on execution shows lots of aborts, however keep running until you get no errors . As the size of the Finite Field is 13, there is no need to run the program that many times, maybe twice.

This program does not sound right as the end-result is a baby Elliptic Curve.

Signing and Verifying a ECC Signature Using Big Values

```
ch1916.py
from sage.all import *
prime = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEC2F
P = FiniteField(prime)
E = EllipticCurve(P, [0, 7])
n = FiniteField(E.cardinality())
print "Cardinality or Number of points on the Elliptic Curve " , E.cardinality()
print "n is not a simple number but a " , n
Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
G = E.point((Gx, Gy))
print "Original Generator Point G on the Elliptic Curve as chosen by us " G
dA = 3
print "Our Private Key d is " , dA
QA = G * dA
print "Our generated Public Key or Point on the Elliptic Curve after EC Multiply is " , QA
e = 4
print "Hash of message e is " , e
print "=====Signing Starts"
k = n.random_element()
if k == 0:
    print "As the random number is 0 e cannot calculate the signature s. Aborting...."
    exit(0)
print "Current random number k generated over the Finite Field is " , k
R = int(k) * G
print "After EC Multiply Generator Point with Random Number. New Point on Elliptic Curve " , R
(x, y, z) = R
if z == 0:
    print "Point at Infinity Aborting...."
    exit(0)
r1 = R.xy()
print "After removing the z coordinate from the New Point " , r1
r = r1[0]
print "After extracting the x value from the New Point the final r value " , r
print "The random number from the Finite Field using r as the index n(r) " , n(r)
modinversek = 1 / k
print "The mod inverse of k is " , modinversek
s = ( 1 / k ) * ( e + n ( r ) * dA)
print "The signature s is " , s
print "The index in n using s n(s)= " , n(s)
if n(s) == 0:
```

```

    print "Aborting as n(s) is 0..."
    exit(0)
print "-----The Final Signature that we compute s " , s , " and r is " , r
print "=====Verifications Starts"
w = 1 / n(s)
print "The mod inverse w using the signature s is " , w
print "w is not a number but a " , type(w)
u1 = int ( w * e ) * G
print "Multiplication of w and e " , w * e
print "First Temp Calculation Point on the EC is " , u1
u2 = int ( n (r) * w ) * QA
print "Multiplication of n(r) and w " , n(r) * w
print "Second Temp Calculation Point on the EC is " , u2
P = ( u1 + u2 )
print "Verification answer or adding the above two points is " , P
r1 = P.xy()[0]
print "-----The value of r after Verification r1 " , r1
print "The two r's are the same " , r == r1

```

Output

```

Cardinality or Number of points on the Elliptic Curve
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
n is not a simple number but a Finite Field of size
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
Original Generator Point G on the Elliptic Curve as chosen by us
(55066263022277343669578718895168534326250603453777594175500187360389116
729240 :
32670510020758816978083085130507043184471273380659243275938904335757337
482424 : 1)
Our Private Key d is 3
Our generated Public Key or Point on the Elliptic Curve after EC Multiply is
(11271166043971060605674865917392967310211497734153940854463061355520977
5888121 :
25583027980570883691656905877401976406448868254816295069919888960541586
679410 : 1)
Hash of message e is 4
=====Signing Starts
Current random number k generated over the Finite Field is
30491088948638766813520597665700963683719444033678797735035545376133648
686834
After EC Multiply Generator Point with Random Number. New Point on Elliptic Curve
(19821846500350087748346076167656058716471595892226550965968787647868105
233516 :
46253253661601103104021637134167104224070219818012126481015480287427401
19944 : 1)

```

After removing the z coordinate from the New Point
(19821846500350087748346076167656058716471595892226550965968787647868105
233516,
46253253661601103104021637134167104224070219818012126481015480287427401
19944)
After extracting the x value from the New Point the final r value
19821846500350087748346076167656058716471595892226550965968787647868105
233516
The random number from the Finite Field using r as the index n(r)
19821846500350087748346076167656058716471595892226550965968787647868105
233516
The mod inverse of k is
51513554941618243777583616273858270796208515696154541115427431210495542
128281
The signature s is
73714230648769410570803725011588708897877015183511227499707461820999379
938386
The index in n using s n(s)=
73714230648769410570803725011588708897877015183511227499707461820999379
938386
-----The Final Signature that we compute s
73714230648769410570803725011588708897877015183511227499707461820999379
938386 and r is
19821846500350087748346076167656058716471595892226550965968787647868105
233516
=====Verifications Starts
The mod inverse w using the signature s is
81298555451819630266162086876862496971815467810660412906869906031293165
367985
w is not a number but a <type 'sage.rings.finite_rings.integer_mod.IntegerMod_gmp'>
Multiplication of w and e
93610043332646130217506377490074172181586742684491842862269297842136338
483266
First Temp Calculation Point on the EC is
(11536063891253714231903685629507806416346301979741049884004127263025606
4503339 :
16761852805648303015798774457504242212089226981429838922003411699628516
763869 : 1)
Multiplication of n(r) and w
56155074696875009147718730064334202402602609969112254545992191272344544
397414
Second Temp Calculation Point on the EC is
(31431097488078659667661855777232958206973487271683315373015564127695835
854333 :
95266117708814924165166986290468554867114005005899516093696259124701652
573201 : 1)

Verification answer or adding the above two points is
 (19821846500350087748346076167656058716471595892226550965968787647868105
 233516 :
 46253253661601103104021637134167104224070219818012126481015480287427401
 19944 : 1)
 -----The value of r after Verification r1
 19821846500350087748346076167656058716471595892226550965968787647868105
 233516
 The two r's are the same True

There are only three changes made to the earlier code.

First, the prime number used is the number used by the secp256k1 standard. The Generator Point G uses the same x and y that the standard demands. The output displays very large numbers though it cannot be confirmed that the points are on an Elliptic Curve visually, like we did earlier. The final answer is True and that is all that counts. This is a working real calculation.

Now to a very basic question. Why these complicated set of calculations? There must be some rational thinking behind all of it.

Let's start with the basics once again.

$$P = u1 + u2$$

This equation finds the Point P. The x coordinate of this point P becomes the value of r1. It must match the value of r generated in the signing process.

We know how to calculate u1 so let's substitute that value in the above equation, but without the Python stuff.

$$P = (e * w * G) + u2$$

The same process is applied for the temporary variable u2 also.

$$P = e * w * G + (r * w * QA)$$

QA is the public key. The private key dA is multiplied with the Generator Point G. Let's substitute it in the above equation.

$$P = e * w * G + r * w * (dA * G)$$

Finally, we get

$$P = e * w * G + r * w * dA * G$$

We see that G is present on both sides of the + sign. So, let's remove G.

$$P = (e * w + r * w * dA) * G$$

What we do for G we can also do for w for the same reasons.

$$P = w * (e + r * dA) * G$$

We know that the value of w is the mod inverse of s.

$$w = 1 / s$$

We calculated s in the signing process as

$$s = (e + dA * r) / k$$

Let's invert both sides of the equation at the same time.

$$1 / s = k / (e + dA * r)$$

We know that $w = 1/s$. We replace $1/s$ with the above value to get a value of w .

$$w = k / (e + dA * r)$$

We now replace the value of w in the P equation above to get

$$P = (k / (e + dA * r)) * (e + dA * r) * G$$

Both the numerator and denominator have $e + dA * r$ so they cancel each other.

$$P = k * G$$

This is how the value of r is created in the signing process. This may look complex so let's look at the whole process through another set of eyes.

Once again, the same equation

$$P = u1 + u2$$

This becomes

$$P = (e * w + r * w * dA) * G$$

w is the mod inverse of s . So, we replace w with s , mod inverse. Done before.

$$P = (e / s + (r * dA) / s) * G$$

Like before, we also move s out from both sides

$$P = (e + r * dA) * G / s$$

So far, a carbon copy or zero copy of what we have done so far.

Let's go back to the original definition of s

$$s = (e + dA * r) / k$$

We will now multiply both sides with k

$$s * k = (e + dA * r) * k / k$$

This will allow us to remove k from the right-hand side of the equation.

$$s * k = (e + dA * r)$$

Let's now divide both sides of the equation with s .

$$s * k / s = (e + dA * r) / s$$

After removing the s on the left-hand side, we get.

$$k = (e + dA * r) / s$$

Let's rewrite the Point P computation from above

$$P = k * G$$

Replacing k, we get

$$P = (e + r * dA) * G / s$$

Just rearranging the equation

$$P = ((e + dA * r) / s) * G$$

The value is replaced with k.

$$P = k * G$$

The result is the same.

What this proves is that despite having only the public key and not the private key, we can verify that the private key signed. The public key is derived from the private key and not the other way around.

Let's change track a bit.

Everyone harps on the fact that r must be random. Let's assume that r is not random but has a known value. For the record, k is a random number. From this k value, we get the point R, from which we get a value of r.

This is how we calculate the signature

$$s = (e + dA * r) / k$$

Let's assume there are two different Bitcoin transactions with the same random number k and hence the same value of r.

The message hashes will be different e1 and e2 and the private key will be the same as the same person signs both messages. We would have preferred m1 and m2 for denoting the messages. The signature is decided by the message data and hence the signatures will be different as s1 and s2.

We get two different equations

$$s1 = (e1 + dA * r) / k$$

$$s2 = (e2 + dA * r) / k$$

We interchange s and k in both equations to get.

$$k = (e1 + dA * r) / s1$$

$$k = (e2 + dA * r) / s2$$

As both sides have the same variable k, they are equal and hence we get

$$(e1 + dA * r) / s1 = (e2 + dA * r) / s2$$

Let's now multiply both sides of the equation with the product s1 * s2

$$(e1 + dA * r) / s1 * s1 * s2 = (e2 + dA * r) / s2 * s1 * s2$$

Let's remove the divisors or the mod inverses on both sides.

$$(e1 + dA * r) * s2 = (e2 + dA * r) * s1$$

Let's now multiply to expand both sides of the equation.

$$e1 * s2 + dA * r * s2 = e2 * s1 + dA * r * s1$$

Subtract $s1 * e2$ from both sides.

$$e1 * s2 + dA * r * s2 - s1 * e2 = e2 * s1 + dA * r * s1 - s1 * e2$$

Tidying up after the subtraction gives us

$$e1 * s2 + dA * r * s2 - s1 * e2 = dA * r * s1$$

Now subtract $s2 * dA * r$ for both sides

$$e1 * s2 + dA * r * s2 - s1 * e2 - s2 * dA * r = dA * r * s1 - s2 * dA * r$$

Tidying up again we get

$$e1 * s2 - e2 * s1 = dA * r * s1 - dA * r * s2$$

There is a constant $dA * r$ on the right-hand side.

$$e1 * s2 - e2 * s1 = dA * r * (s1 - s2)$$

The right-hand side has the private key so we divide both sides with $r * (s1 - s2)$

$$(e1 * s2 - e2 * s1) / (r * (s1 - s2)) = dA * r * (s1 - s2) / (r * (s1 - s2))$$

After removing the common values from the denominator and numerator, we get.

$$(e1 * s2 - e2 * s1) / (r * (s1 - s2)) = dA$$

Reversing the left and the right side. This will not change the value.

$$dA = (e1 * s2 - e2 * s1) / (r * (s1 - s2))$$

This process concludes that if we have the two signatures, $s1$ and $s2$ and the two different message hashes, $e1$ and $e2$ and a constant random number r , the private key dA can be figured. But prior to that, how about also figuring out the random number used. We have the value of r and not k .

Let's start once again with the two equations as above.

$$s1 = (e1 + dA * r) / k$$

$$s2 = (e2 + dA * r) / k$$

Let's expand both.

$$s1 = e1 / k + dA * r / k$$

$$s2 = e2 / k + dA * r / k$$

Let's move the messages to the left-hand side from the right-hand side

$$s1 - e1 / k = dA * r / k$$

$$s2 - e2 / k = dA * r / k$$

You realize that the right-hand sides are the same, so we get

$$s_1 - e_1 / k = s_2 - e_2 / k$$

Let's get the signatures on the left-hand side and the messages on the right-hand sides.

$$s_1 - s_2 = e_1 / k - e_2 / k$$

Rearranging

$$s_1 - s_2 = (e_1 - e_2) / k$$

We will now multiply both sides with k

$$k * (s_1 - s_2) = (e_1 - e_2) / k * k$$

Removing the common k from the right-hand side

$$k * (s_1 - s_2) = (e_1 - e_2)$$

We now divide each side by (s1 - s2)

$$k * (s_1 - s_2) / (s_1 - s_2) = (e_1 - e_2) / (s_1 - s_2)$$

Removing the same (s1 - s2) from the left-hand side.

$$k = (e_1 - e_2) / (s_1 - s_2)$$

Let's test out whether the above equations leak out the private key dA and the random number k.

Obtaining the private key by signing two messages

```
ch1917.py
from sage.all import *
P = FiniteField(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)
E = EllipticCurve(P, [0,7])
Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
G = E.point((Gx, Gy))
n = FiniteField(E.cardinality())
dA = 12345678901234567890123456789012345678901234567890
QA = G * dA
e = 2
e1 = e
print "First Message ", e
k = n.random_element()
tmp = (int(k)) * G
r1 = tmp.xy()
r = (tmp).xy()[0]
r1 = r
print "First r ", r
s = (1 / k) * (e + n(r) * dA)
s1 = s
```

```
print "First Signature s " , s
w = 1 / n ( s )
tmp = int ( w * e ) * G
tmp1 = int ( n (r) * w ) * QA
ansf = ( tmp + tmp1 ).xy()[0]
print "First Verification is " , r == ansf
print
#Finding out the Private Key
e = 3
e2 = e
print "First Message " , e2
tmp = (int (k)) * G
r1 = tmp.xy()
r = (tmp).xy()[0]
r2 = r
print "Second r " , r2
s = ( 1 / k ) * ( e2 + n (r) * dA)
s2 = s
print "Second Signature s " , s2
w = 1 / n ( s2 )
tmp = int ( w * e2 ) * G
tmp1 = int ( n (r) * w ) * QA
ansf1 = ( tmp + tmp1 ).xy()[0]
print "Second Verification is " , r == ansf1
print
tmp2 = s2 * e1 - s1 * e2
tmp3 = int(r) * (s1 - s2)
dAfinal = tmp2 / tmp3
print "Our original private Key dA is " , dA
print "The private key calculated is " , dAfinal
#Finding out k used
k = ( e1 - e2 ) / ( s1 - s2)
print
print "The random number k calculated is " , k
print "The random number we used is " , k
```

Output

First Message 2

First r 23151278288765770733927708169037044425098709266397014844277324852752534
63852

First Signature s

49901924387301531857067247933437167088475402689690140977656122645974447
860952

First Verification is True

First Message 3

Second r

```

23151278288765770733927708169037044425098709266397014844277324852752534
63852
Second Signature s
60641386415120315251986414515061218167617453599506254960540941049181912
730313
Second Verification is True

Our original private Key dA is
12345678901234567890123456789012345678901234567890
The private key calculated is
12345678901234567890123456789012345678901234567890

The random number k calculated is
79435449493288158554556599553955217262363110063454523738853914673158356
500552
The random number we used is
79435449493288158554556599553955217262363110063454523738853914673158356
500552

```

This example effectively brings in nothing new to the party.

A very large private key d_A and the same parameters for the secp256k1 Elliptic Curve are used. The message e has a value, 2. The same message value is saved in a variable $e1$ for later use. A random number k is used a little later to calculate the signature for another message, $e2$. Then the value of r and s are calculated using r and k and the message, $e1$. This signature is stored in the $s1$ variable for later use. There is no real need to verify anything but it's been done for abundant caution.

Now to the second message $e2$ whose hash is 3. The value of r is re-calculated and it has the same value as the random number remains the same. This value is saved in $r2$ even though it is not used as r , $r1$ and $r2$ will be the same. However, the signature $s2$ changes as the message has changed.

The formula applied earlier is used to calculate the private key. So, first multiply $s2 * e1 - s1 * e2$. The result is stored in a variable called $tmp2$. Then find out the result of $r * (s1 - s2)$ and save it in variable $tmp3$. Then ask Sage to divide $tmp2$ by $tmp3$. The same private key is what we are left with. To find the value of k is much simpler, once again the inverse mod is calculated by Sage. Both private key and random number match to the T .

All that this example proves is that if we, by mistake use the same random number to sign two Bitcoin transactions then our private key will not be leaked but it will be publicly known. This happened to us.

CHAPTER 20

ECC with Sage - Part 2

In this chapter, we look at some advanced features of Sage and then understand them with pure Python code.

Lift Gets a y Coordinate on an Elliptic Curve, given a x Coordinate

```
ch2001.py
from sage.all import *
P = FiniteField(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC2F)
E = EllipticCurve(P, [0,7])
Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
G = E.point((Gx, Gy))
G1 = E.lift_x(0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798)
print G1
print "%x" % G1[1]
print "%x" % Gy
```

```
Output
(55066263022277343669578718895168534326250603453777594175500187360389116
729240 :
32670510020758816978083085130507043184471273380659243275938904335757337
482424 : 1)
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
```

The first part of the above program has been explained earlier. A secp256k1 Elliptic Curve is created and a Generator Point G is defined on it. Most of the time, there will be an x coordinate and no y coordinate for the point. The function `lift_x` takes a single parameter, which is an x coordinate. A point is returned with the same x value and a corresponding y value.

The output displays G1 as a point on the Elliptic Curve and the returned y coordinate and the original y coordinate. They both are the same. Thus, if an x value on the Elliptic Curve is available, Sage will figure out the y value for us.

The Difference Between Order and Cardinality

```
ch2002.py
from sage.all import *
F = FiniteField(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC2F)
```

```

E = EllipticCurve(F , [0 , 7])
Gx = 55066263022277343669578718895168534326250603453777594175500187360389116
729240
Gy = 32670510020758816978083085130507043184471273380659243275938904335757337
482424
G = E.point((Gx , Gy) )
print E.order()
print E.cardinality()
print "Cardinality == Order for the Elliptic Curve " , E.order() == E.cardinality()
print G.order()
print "G.order() == E.order() " , E.order() == G.order()
#print G.cardinality()
N=0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
print N
print "N == G.order() " , G.order() == N

```

Output

```

11579208923731619542357098500868790785283756427907490438260516314151816
1494337
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
Cardinality == Order for the Elliptic Curve True
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
G.order() == E.order() True
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
N == G.order() True

```

The difference between cardinality and order is explained one step at a time. All the Elliptic Curves use the secp256k1 parameters, unless otherwise stated. Like before, both the cardinality and order of an Elliptic curve are displayed and they are the same. These are standard functions in the Sage module. G is our standard point on the Elliptic Curve. It also has an order, which is like the order of an Elliptic curve.

However, a point does not have a cardinality function. Hence the comment. This should be obvious as cardinality talks about count as in how many. A point is only one member or entity. So, the concept of cardinality does not apply to points but to curves only. Try and display the cardinality of a point and Sage throws an exception. The order of a point can be defined as the number of times one must 'circle back' before coming back to the start. This is also equal to the number of points on the Elliptic Curve.

Let's remind you again that the n or N variable used by the secp256k1 standard represents the number of points on the curve. It is all about the order. The hardcoded value of N is used since the value is entrusted by the standard or to be precise, the designers of the Elliptic Curve.

Sage and Prime Numbers

```

ch2003.py
from sage.all import *
print primes_first_n(5)

```

```
print 5 in Primes()
print 6 in Primes()
```

Output

```
[2, 3, 5, 7, 11]
True
False
```

Sage excels at several things, one of them is prime numbers. It has a global function called `primes_first_n`. The function is given a number, 5 so it returns the first 5 primes. More importantly is the function `Primes` that returns `True` if the number before the `in` clause is a Prime number. 5 is a prime number so the return value is `True`. For 6, a value of `False` is returned. One more nail in the coffin to use Sage.

The Order and Cardinality are Not the Same Always

```
ch2004.py
from sage.all import *
prime = 9
while prime <= 17:
    prime = prime + 1
    if not prime in Primes():
        continue
    K = FiniteField(prime)
    E = EllipticCurve(K , [0 , 7])
    print "(%02d) Order of Curve is %02d:%d" % (prime , E.order() , E.cardinality())
    for x in range (prime):
        for y in range (prime):
            lhs = (y * y) % (prime )
            rhs = ((x * x * x) + 7) % (prime )
            if lhs == rhs:
                G = E([x , y])
                print "\t(%02d,%02d) order %02d" % (x , y , G.order())
```

Output

```
(11) Order of Curve is 12:12
      (02,02) order 04
      (02,09) order 04
      (03,01) order 03
      (03,10) order 03
      (04,04) order 12
      (04,07) order 12
      (05,00) order 02
      (06,05) order 06
      (06,06) order 06
      (07,03) order 12
      (07,08) order 12
(13) Order of Curve is 07:7
      (07,05) order 07
```

```

(07,08) order 07
(08,05) order 07
(08,08) order 07
(11,05) order 07
(11,08) order 07
(17) Order of Curve is 18:18
(01,05) order 09
(01,12) order 09
(02,07) order 09
(02,10) order 09
(03,00) order 02
(05,08) order 03
(05,09) order 03
(06,06) order 18
(06,11) order 18
(08,03) order 06
(08,14) order 06
(10,02) order 18
(10,15) order 18
(12,01) order 09
(12,16) order 09
(15,04) order 18
(15,13) order 18

```

There can never be a single and simple explanation to understand something as complex as Elliptic Curves. With Sage, life is easy. It can be made part and parcel of Python.

The variable prime is initialized to 9, a non-prime number, on purpose. In the outer while loop, it is increased by 1 so its value is now 10. As 10 is not a prime number, it loops back to the start.

The Primes function ensures we are dealing with prime numbers only in the while loop. The FiniteField K is created with the size of the new prime number and an Elliptic Curve, E.

There are two for loops, one for the x coordinate and one for the y coordinate, both loops stop on reaching a prime number value.

The task is to create a Point on the Elliptic curve using the variables x and y. Generally, the code must be in a try clause but we take a rain check today. At first the left-hand side is calculated and then separately, the right side of the equation. The equation is the one used to create a secp256k1 curve. Now, that modulo arithmetic is outside of Sage, the modulus operator % comes in. The if statement checks if the two sides are equal, if so then we have a valid point on the curve. The order and not the cardinality of that point is displayed.

There are Two Points on the Curve for Every x Coordinate

```

ch2005.py
from sage.all import *
prime = 11
K = FiniteField(prime)
E = EllipticCurve(K, [0,7])
for x in range (prime):

```

```
for y in range (prime):
    lhs = (y * y) % (prime )
    rhs = ((x * x * x) + 7) % (prime )
    if lhs == rhs:
        print “(%02d , %02d) (%02d:%02d)” % (x , y , x , (-y) % prime) ,
        G = E.point((x,(-y) % prime) )
        print G
```

Output

```
(02 , 02) (02:09) (2 : 9 : 1)
(02 , 09) (02:02) (2 : 2 : 1)
(03 , 01) (03:10) (3 : 10 : 1)
(03 , 10) (03:01) (3 : 1 : 1)
(04 , 04) (04:07) (4 : 7 : 1)
(04 , 07) (04:04) (4 : 4 : 1)
(05 , 00) (05:00) (5 : 0 : 1)
(06 , 05) (06:06) (6 : 6 : 1)
(06 , 06) (06:05) (6 : 5 : 1)
(07 , 03) (07:08) (7 : 8 : 1)
(07 , 08) (07:03) (7 : 3 : 1)
```

The Elliptic curve is called a curve for a reason. For every x on the curve, there are two y values. One is the actual value of y and the other is the reflected one, which is -y.

Let's look at the above example for inspiration. The prime number used is 11. The logic is same as before where a point (x, y) is located on the curve. A calculation of -y % prime is performed to get the other reflected value, which is also a point on the Elliptic Curve. The reflected value is then displayed. The two consecutive rows dance to the same tune.

It is also confirmed that by creating a point using the reflected value -y on their curve, no error is reported. However, when the value of y is 0, its reflected value is also 0.

If there is an Addition, then there is No Need for a Multiplication

```
ch2006.py
from sage.all import *
prime = 11
K = FiniteField(prime)
E = EllipticCurve(K , [0,7])
G = E([3 , 1])
G = G * 2
print “(%d,%d)” % (G[0] , G[1])
G1 = E([3 , 1])
G1 = G1 + G1
print “(%d,%d)” % (G1[0] , G1[1])
print G == G1
```

Output

```
(3,10)
(3,10)
True
```


As mentioned earlier, the Elliptic Curve multiplication is very different from the mathematical multiplication. ECC multiplication gives us another point on the Elliptic Curve.

The same logic is applied to Adding two points. In this example, a point G is multiplied by 2 thus resulting in the same answer as adding two points. Multiplication is nothing but repetitive addition everywhere.

Multiplying a Point by the Order Gives a Point at Infinity

```
ch2007.py
from sage.all import *
prime = 11
K = FiniteField(prime)
E = EllipticCurve(K, [0,7])
G = E([3, 1])
G = G * E.order()
print G
G = E([3, 1])
G = G * (E.order() + 1)
print G
```

```
Output
(0 : 1 : 0)
(3 : 1 : 1)
```

A simple recap. When any point is multiplied by the order, the result is an identity point or the point at infinity, the z coordinate is 0 here. At the same time, multiplying a point by its order plus 1 returns to the same point again. A circle has been crossed.

Addition in ECC is Both Commutative and Associative

```
ch2008.py
from sage.all import *
K = FiniteField(2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1)
E = EllipticCurve(K, [0,7])
print E(0)
Gx = 55066263022277343669578718895168534326250603453777594175500187360389116
729240
Gy = 32670510020758816978083085130507043184471273380659243275938904335757337
482424
G = E([Gx, Gy])
G1 = G + E(0)
print G == G1
G2 = E.lift_x(4)
print G + G2 == G2 + G
G2 = E.lift_x(1)
print G + (G1 + G2) == (G + G1) + G2
print 3 * G == G + G + G
```

```
Output
(0 : 1 : 0)
```

True
True
True
True

The first element or the identity element is $E(0)$. This is the actual secp256k1 curve. G is the Generator Point. When the identity element is added to this point, the same Generator Point is achieved. There is nothing special about it. The point to be noted is that adding an identity point to a point does not generate a new point, it remains the same. It is like addition with a 0.

The point G_2 is another Point on the Elliptic Curve.

Addition has a property called Commutative which means that $4 + 2 = 2 + 4$. A big word which impresses non-mathematicians. The next example brings in another big math word, addition is also associative which means that $(2 + 3) + 4 = 2 + (3 + 4)$.

As mentioned earlier, $3 * G = G + G + G$.

It does not matter whether the addition is in form of, $G + G_2$ or $G_2 + G$. The answer is the same. Finally, $G + (G_1 + G_2)$ is the same as $(G + G_1) + G_2$. Thus, ECC addition is Commutative and Associative.

Mathematicians and cryptographers do not trust numbers. They look at every number used with suspicion and doubt. as they believe it is some backdoor entry to weaken the cryptography.

There is an ongoing debate on the Bitcoin forums which question the chosen constants being random numbers. Some of the Sage code in the book are from those discussions. We have been inspired by a trillion kindred souls.

Prime Numbers and the Constant Chosen are the Same

```
ch2009.py
from sage.all import *
Pcurve1 = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
for t in range(1024, -1, -1):
    Pcurve = 2**256 - 2**32 - t
    if Pcurve in Primes():
        print "Prime Number found at t %d" % t
        print Pcurve
        if Pcurve == Pcurve1:
            print "Prime Numbers are the same as constant chosen"
            break
    else:
        print "Vijay Mukhi is an embecile"
```

Output

```
Prime Number found at t 977
11579208923731619542357098500868790785326998466564056403945758400790883
4671663
```

The prime number used in standard secp256k1 is a random number. The first suspicious number is this prime number. The variable `Pcurve1` stores the original prime number used by the secp256k1 standard.

In the for loop, `t` starts with 1024 but it reduces by 1 in the loop as the last parameter is -1.

It is assumed that the prime number to be used has a base of $2^{256} - 2^{32}$, which looks like a random number. The variable Pcurve is assigned this prime number.

The value of t is reduced from this large number, Pcurve which takes values like 1024, 1023, 1022 etc.

A check is performed on the resultant number Pcurve. It must be a prime number. If so, then the value of t is printed. In our case, variable t has a value of 977. The prime number happens to be same as the one used by the secp256k1 standard.

This also proves that the prime number used by the secp256k1 standard is the smallest prime number after $2^{256} - 2^{32}$.

An Explanation of the Value 7 Assigned to Constant b

```
ch2010.py
from sage.all import *
p = 11579208923731619542357098500868790785326998466564056403945758400790883
4671663
K = FiniteField(p)
for b in range(1,10): # we start at 1 not 0 because 0 is a singular curve
    E = EllipticCurve(K, [0, b])
    if E.order() in Primes():
        print "Curve Found at a 0 b %d" % b
        print "p prime %d" % p
        print "Order is %d" % E.order()
        exit(0)
```

Output

```
Curve Found at a 0 b 7
p prime
11579208923731619542357098500868790785326998466564056403945758400790883
4671663
Order is
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
```

The next riddle is on the chosen constant value, 7. The same prime number is verified as a random number.

The value of variable b starts from 1 and not 0 because 0 gives an exception of having defined a singular curve. Singular curves, we don't understand.

In the for loop, multiple Elliptic Curves are created with b having values of 1, 2, 3 etc. Thereafter, the task is to figure out the order of the Elliptic Curve or count the points it has. The Order function must return a prime number. The Primes function determines whether the newly created Elliptic Curve is a prime or not, based on the number of points on the it. The first value of b where this holds true is 7 and hence the secp256k1 standard chooses the value 7 for the variable b.

When the order of an elliptic curve is a prime number, it is called an elliptic curve of prime order.

Why was the Value of Constant a Chosen to be 0?

```
ch2011.py
from sage.all import *
```

```
p = 11579208923731619542357098500868790785326998466564056403945758400790883
4671663
K = FiniteField(p)
for a in range ( 0 , 10):
    for b in range(1,10): # we start at 1 not 0 because 0 is a singular curve
        E = EllipticCurve(K , [a , b])
        if E.order () in Primes():
            print "Curve Found at a %d b %d" % ( a , b)
            print "p prime %d" % p
            print "Order is %d" % E.order()
            exit(0)
```

Output

```
Curve Found at a 0 b 7
p prime
11579208923731619542357098500868790785326998466564056403945758400790883
4671663
Order is
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
```

The next question is why 0 for the constant a? Was this random or was it a backdoor? The same program now has an extra for loop where the variable a starts from 0. An Elliptic Curve is created by changing both the values of a and b. The first for loop is not needed as a is 0.

Therefore, the designers of secp256k1 standard chose a and b as 0 and 7 respectively. These were the smallest values which give a prime order Elliptic Curve, wherein the order of the curve or number of points on the curve is a prime number.

```
ch2012.py
from bitcoin import *
Gx = 55066263022277343669578718895168534326250603453777594175500187360389116
729240
Gy = 32670510020758816978083085130507043184471273380659243275938904335757337
482424

pub = privtopub(1)
print "%x" % pub[0]
print "%x" % Gx
print "%x" % pub[1]
print "%x" % Gy
if (Gx == pub[0]) and (Gy == pub[1]):
    print "All is good"
else:
    print "Vijay Mukhi is an embecile"

>python ch2001.py
```

Output

```
79be667ef9dcbbac55a06295ce870b07029bfcd2dce28d959f2815b16f81798
```

```

79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
All is good.

```

This example repeats most of the things learnt so far. The public key is the private key multiplied by the generator point and a Generator point is a tuple of two values, x and y.

The Generator Point used is the one used by secp256k1. There is a lot of research that proves that the Generator Point can be any value/point on the Elliptic Curve. The value chosen does not affect the security of the Elliptic Curve Cryptography in any way.

The `privtopub` function from the bitcoin library computes the public key. A value of 1 as private key given to this function retrieves the original Generator Point G.

So far, Sage was used for the inverse mod and the Elliptic Curve multiplication and addition. We now attempt to do Elliptic Math ourselves.

A Factor and a gcd

```

ch2013.py
def factor1(x):
    l= []
    for i in range(1, x + 1):
        if x % i == 0:
            l.append(i)
    return l

n1 = 16
n2 = 40
a = factor1(n1)
print a
b = factor1(n2)
print b
c = list(set(a).intersection(b))
print c
print "GCD is %d" % max(c)
print "Number of items is %d" % len(c)

```

Output

```

[1, 2, 4, 8, 16]
[1, 2, 4, 5, 8, 10, 20, 40]
[8, 1, 2, 4]
GCD is 8
Number of items is 4

```

The program has two variables `n1` and `n2` with the values of 16 and 40. Take them as random numbers. Let's now factor these numbers, what they do is not important, for now.

We are giving a number, 16 as the `x` value to the `factor1` function. The function starts with an empty list called `l`. There is a for loop which starts from 1 and ends at `x + 1`.

Then we have the modulus and not the divisor. The two values supplied to the modulus are from parameter x and the loop variable i. The value in x remains constant throughout and the variable i starts from 1 and ends at the x value. If the number is divisible by some number, the remainder or modulus is 0.

A factor is found when a number mod(%) by another gives a 0. It indicates that a factor is found for the number. A mod of 3 by 2 gives a remainder of 1, which means that 2 is not a factor of 3. However, 4 mod 2 gives a 0, so 2 is a factor of 4.

When a factor is found, the value of variable i is appended to the list l using the append function. When the for loop ends, the list l has all the factors.

There are two list variables, list a contains the factors of 16 and list b contains the factors of 40. The number itself and 1 will always be a factor. These factors are displayed on the screen.

From the two lists, the common numbers or factors are obtained. A simple method here is to convert list a into a set. This is performed using the set constructor. The intersection function of the set class is called by passing it the second list b.

The variable c now contains the common members of both lists which are 8, 1, 2 and 4. The max function returns the maximum value in the list which is 8. This number is called the GCD or the Greatest Common Divisor.

Once again, a factor is a number that completely divides a number leaving a remainder of 0. So, we find all the factors of a number. The largest factor common to both lists is the GCD.

```
n1 = 13
```

A single change incorporated to our program where the variable n1 now has a value of 13. The result is as follows:

```
[1, 13]
[1, 2, 4, 5, 8, 10, 20, 40]
[1]
GCD is 1
Number of items is 1
```

The only factor these two numbers 13 and 40 have in common is 1.

Many More Ways to Find gcd

```
ch2014.py
def gcd(a,b):
    while b:
        dummy = a % b
        print "gcd start dummy %d a %d b %d " % (dummy , a , b)
        a = b
        b = dummy
        print "gcd at end dummy %d a %d b %d " % (dummy , a , b)
    return a

def gcd1(a,b):
    while b:
        a,b = b, a % b
        print "gcd1 a %d b %d" % (a , b)
    return a
```

```

def gcd2(a, b):
    while a != b:
        print "gcd2 Start a %d b %d" % (a , b)
        if a > b:
            a = a - b
        else:
            b = b - a
        print "gcd2 End a %d b %d" % (a , b)
    return a

def gcdr(a,b):
    print "gcdr a %d b %d" % (a , b)
    if b == 0:
        return a
    else:
        return gcdr(b, a % b)

a = gcd(16,40)
print "ans %d\n" % a
a = gcd1(16,40)
print "ans %d\n" % a
a = gcd2(16,40)
print "ans %d\n" % a
a = gcdr(16,40)
print "ans %d\n" % a

```

Output

```

gcd start dummy 16 a 16 b 40
gcd at end dummy 16 a 40 b 16
gcd start dummy 8 a 40 b 16
gcd at end dummy 8 a 16 b 8
gcd start dummy 0 a 16 b 8
gcd at end dummy 0 a 8 b 0
ans 8

gcd1 a 40 b 16
gcd1 a 16 b 8
gcd1 a 8 b 0
ans 8

gcd2 Start a 16 b 40
gcd2 End a 16 b 24
gcd2 Start a 16 b 24
gcd2 End a 16 b 8
gcd2 Start a 16 b 8
gcd2 End a 8 b 8
ans 8

gcdr a 16 b 40
gcdr a 40 b 16

```

```
gcdr a 16 b 8
gcdr a 8 b 0
ans 8
```

This program has not one but four different ways to calculate the GCD. There is a function for each of them and each function will calculate the largest common divisor of two numbers. The GCD is calculated of numbers 16 and 40. The answer is 8.

Let's start with the first function called gcd. The while loop continues till the value of b becomes 0 from 40. First, a mod value is obtained of the first number 16 with the second number 40.

In mod, when the first value is less than the second value, the answer is the first number. Using actual values, $16 \% 40$ gives 16 as the remainder even though they have factors in common. The modulus operator % however does not understand factors.

The value of dummy after the first modulus operation is 16. Then a is assigned the value of b, thus the value in a is deleted. Thereafter, the value of b is changed to that of dummy which is 16.

At the start of the second iteration, the formula is $40 \% 16$. Now the remainder is 8 as 40 is divisible by 16 and the divisor is smaller. The value of variable a now becomes 16 (the value of b), and b is 8, (the value of dummy). The variable b keeps reducing each time in the loop. In the last round, the remainder will be 0 as 16 divided by 8 is 0. The value of b is now the value of dummy which is 0.

The value in variable b is reduced until the remainder is 0. This is the largest common divisor between the two numbers. The idea is to keep modding the two numbers, reducing the second one until there is no remainder or the remainder is 0. This is the GCD. At some point in time, the value of b will be 1, which divides every number.

The second function gcd1 is like the first but there is no dummy variable. The variables a and b are changed with b and $a \% b$ in one line.

The third function gcd2 however uses a different approach. Here, the loop continues till the two numbers meet a certain condition. They must be different.

The if statement will be called if and only if variable a is greater than variable b, otherwise the else is called. In the else clause, the variable b is reduced by variable a, so now the value of variable b reduces. In the if statement, the value in variable a is reduced by the value in variable b. The value of variable b reduces from 40 to 24.

In the second iteration of the while loop, b is still greater than a, so its value is reduced by giving it a value of 8. Now value in a, 16 is larger than b, 8. So, the if statement kicks in. Here the value of the variable a is reduced by the value in b, 8. Both a and b, have the same value, so either of them can be returned.

Finally, recursion is implemented in the last GCD function, gcdr. the variable a is set to b and b to the modulus of $a \% b$. This keeps going on. It stops when variable b becomes 0. We prefer the recursive approach because the code is concise and clean.

Change variable a to 13, the gcd is 1.

The Modulo Inverse of a Number

```
ch2015.py
a = 13
field = 17
for i in range(1, 20):
    b = (a * i) % field
```



```

print i * a , b
if b == 1:
    print "Found a inverse at i %d" % i
    break

```

Output

```

13 13
26 9
39 5
52 1
Found a inverse at i 4

```

Let's now calculate the modulo inverse of a number 13 over a finite field, in this case a simple number. A modular inverse is defined over a finite field, in our case that field is the prime number 17.

The task is to calculate another number which when multiplied by 13 results in a value of 1. The prime number 17 is used to find the remainder of this multiplication. Sage performs this task in the background.

In the program, the variable a has the value who's modulo inverse is to be determined. The variable field is the finite field size, 17.

The loop starts at 1 and goes on till an arbitrary chosen value of 20. The loop variable is called i. In the loop, the value in variable a is multiplied by i and then the remainder or modulus is determined. When i is 1, the multiplication results in 13. 13 modded by 17 is 13. In the next iteration of the for loop, the i variable has a value of 2, the multiplication is 26, the remainder is 26 % 17 that results in 9. In the fourth iteration, i is 4, the multiplication is 52. 52 divided by 17 gives an answer of 3 and remainder of 1. Thus, this value of i which is 4 is the modular inverse of 13.

This method is too slow for large numbers, so let's use recursion to achieve the same result.

A Faster Way of Computing the Mod Inverse

ch2016.py

```

j = 0
def egcd(a, b):
    global j
    j = j + 1
    print "(%d) Start a %d b %d " % ( j , a , b ) ,
    if a == 0:
        print "\n(%d) In first return b %d" % (j , b)
        return (b, 0, 1)
    print " b %% a %d new a %d new b %d" % ( b % a , b % a , a)
    gcd, y, x = egcd(b % a, a)
    print "(%d) End (b %% a) %d x %d gcd %d calc/x %d y %d" % ( j , (b % a) , x , gcd , x - (b // a) * y , y )
    j = j + 1
    return (gcd, x - (b // a) * y, y)

def modinv(a, m):
    gcd, x, y = egcd(a, m)
    print "gcd %d x %d y %d " % (gcd , x , y) ,
    if gcd != 1:
        print "—Numbers are not coprime"

```

```
    return None
else:
    print "—Numbers are coprime %d" % (x % m)
    return x % m

print 10//3
print -10//3

a = modinv(13, 17)
print "The modulo inverse is %d" % a
```

Output

```
3
-4
(1) Start a 13 b 17 b % a 4 new a 4 new b 13
(2) Start a 4 b 13 b % a 1 new a 1 new b 4
(3) Start a 1 b 4 b % a 0 new a 0 new b 1
(4) Start a 0 b 1
(4) In first return b 1
(4) End (b % a) 0 x 1 gcd 1 calc/x 1 y 0
(5) End (b % a) 1 x 0 gcd 1 calc/x -3 y 1
(6) End (b % a) 4 x 1 gcd 1 calc/x 4 y -3
gcd 1 x 4 y -3 —Numbers are coprime 4
The modulo inverse is 4
```

Let's now look at other ways of calculating the modulo inverse of a number. There is one Python operator, `//`, also called the floor divisor.

Dividing 10 by 3 gives 3.333. With the floor divisor, the .333 after the decimal point is dropped, so the result is 3. If one of the numbers is negative, then the floor of the number is used i.e. the answer is -4 and not -3. Remember floor means lower value and -4 is less than -3.

The function `modinv` simply takes two parameters the number to be mod inverted, i.e. 13 and the finite field size 17. In the `modinv` function, first the function `egcd` is called which returns a triplet as the answer. The mod inverse is the mod of the second parameter `a` with `m` the prime field.

The first parameter returned by the function `egcd` is the GCD. This value must be 1 if the modulus inverse is valid. Two numbers are defined as coprime if they have a common divisor of 1 and there is no other factors in common. The last parameter `y` is used later in another program.

The workhorse is the recursive function `egcd`. Generally, in recursion, the recursive function is always called at the end. This one has the recursion function in the middle.

To understand this sort of recursion, a global variable `j` is created and initialized to 0. To make the variable global, the `global` keyword is used. The global variable is used in function `egcd` to retain its value, otherwise it becomes a local variable and it will be initialized during every recursive call. The value of `j` is incremented both at the beginning and at the end of the `egcd` function call for a very good reason.

To start with, the value of global variable `j` is 1, and `a` is 13 and `b` is 17. As the variable called `a` has a non-zero value, the `if` statement will not be called for some time, so ignore it.

In the last example, the variable `b` was changed, here the variable `a` is changed. `b % a` gives 4 and the `egcd` function is called once again, with the new values.

This recursion increments j by 1 again and the process continues; the value of variable a is reduced by executing $b \% a$ and calling the `egcd` function again.

When $b \% a$ results in 1, the `egcd` function is called with variable j which has a value 3. As a is 1 the mod is 0, the `egcd` function is called the last time. The if statement finally gets executed and the value in b is returned, which is 1 and 0 and 1.

Everything now happens backwards; the last value of j is 4 and the value returned by function `egcd` is 1 0 1. This makes GCD as 1 and y as 0 and x as 1.

The value in j is incremented by 1 again, so its value is 5. This value is not the one returned in the `modinv` function. There are many more levels of recursion to cross, so go figure.

The heart of the program is the complex calculation performed to obtain the mod inverse. The current value of x is taken and replaced with $x - \text{the floor division of } b \text{ with } a \text{ and } a \text{ multiplication of } y$.

Let's now understand the modulo inverse calculation without recursion.

A Better Way to Compute the Mod Inverse

ch2017.py

```
def eea(a, prime):
    (s, t, u, v) = (1, 0, 0, 1)
    while prime != 0:
        #print "prime %d" % prime
        (unew, modinverse) = (s, t)
        s = u - (a // prime * s)
        t = v - (a // prime * t)
        (a, prime) = (prime, a % prime)
        v = modinverse
        u = unew
    return modinverse

def modinv1(a, prime):
    pmodinverse = 0
    modinverse = 1
    prevloop = prime
    loop = a % prime
    print "a %d prime %d loop %d prevloop %d\n" % (a, prime, loop, prevloop)
    while loop > 1:
        ratio = prevloop / loop
        print "loop=%d high %d ratio %d" % (loop, prevloop, ratio)
        nm = pmodinverse - modinverse * ratio
        new = prevloop - loop * ratio
        print "nm %d new=%d" % (nm, new)
        prevloop = loop
        pmodinverse = modinverse
        loop = new
        modinverse = nm
    print "loop=%d modinverse %d pmodinverse %d prevloop %d\n" % (loop, modinverse,
        pmodinverse, prevloop)
```

```
        return modinverse % prime
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    g, y, x = egcd(b % a, a)
    return (g, x - (b // a) * y, y)
def modinv2(a, m):
    gcd, x, y = egcd(a, m)
    if gcd != 1:
        return None
    else:
        return x % m
a = modinv1(13, 17)
print "Mod Inverse is %d" % a
if a != 4:
    print "Vijay Mukhi is an embecile"
a = modinv2(13, 17)
print "Mod Inverse is %d" % a
modinverse = eea(13, 17)
print "Mod Inverse is %d" % (modinverse)
```

Output

```
a 13 prime 17 loop 13 prevloop 17
loop=13 high 17 ratio 1
nm -1 new=4
loop=4 modinverse -1 pmodinverse 1 prevloop 13
loop=4 high 13 ratio 3
nm 4 new=1
loop=1 modinverse 4 pmodinverse -1 prevloop 4
Mod Inverse is 4
Mod Inverse is 4
Mod Inverse is 4
```

The method used in the above program calculates the mod inverse of a number in a simpler way, so it will be used henceforward.

The two parameters passed to the modinv1 function are : a with a value of 13 and prime having a value of 17. These values do not change in the function.

The loop variable is called loop. It starts with a value of 13 and quits at 1.

The variable modinverse stores the mod inverse of number. The variable pmodinverse contains the previous value of modinverse. The variable prevloop follows the loop, it stores the previous value of the loop variable before the value changes.

The trick in understanding this code is deciphering and computing the value of the modinverse variable. It has a value of 1 at the beginning of the code. The value of the variable ratio is also 1, it then becomes 3. The nm variable gets its value from the modinverse variable. It starts with -1 and then 4, which is the return value.

Once again, a very poor explanation of the code. We decided to give up explaining this sort of equations in mathematics. Not our finest hour. We wanted to remove the last couple of programs as we could not explain them properly and also, they are not germane to understanding what comes next. But, we tried.

Calculating the Slope of a Line

```
ch2018.py
i = ( 1 , 3 , 6 , 8)
for y in i:
    x = 2 * y + 3
    print "(%d , %d)" % (x , y)
s1 = (3.0-1)/(9.0-5)
print s1
s1 = (6.0-3)/(15.0-9)
print s1
s1 = (8.0 - 2)/(19.0 - 5)
print s1
```

Output

```
(5 , 1)
(9 , 3)
(15 , 6)
(19 , 8)
0.5
0.5
0.428571428571
```

This program looks at the slope of a line. There is a simple line that is denoted by the equation :

$$x = 2 * y + 3$$

Everything is random about this line. The program has a for loop to generate 4 points on the line. The list i represents the y coordinates, the x coordinates are to be determined. Using the above simple equation, the values for y are substituted and in return, an x value is obtained. Finally, the 4 points are displayed. No rocket science here.

The slope of a line is drawn from this equation :

$$m = dy/dx = (y2 - y1) / (x2 - x1) .$$

The values of y2 and y1 are 3 and 1 and the corresponding x2 and x1 values are 9 and 5. A slope is to be measured between points 2 and 3. This slope remains the same for two different set of points.

In the third case, the slope is different from the first because the chosen points are not on the line.

The slope is represented by the derivate or the rate of change dy/dx, the variable m comes handy. For two points, a line is drawn between them but for a single point, a tangent line is drawn at that point. This becomes the slope of the line with two sets of coordinates.

When dealing with the slope of a single point, two points or their difference is not required. The mathematicians call it an infinitesimal slope dx/dy.

What follows is a layman's understanding of the mathematics behind ECC. Do not take us to the cleaners as we do not have the mathematics gene in us. Can be skipped.

The secp256k1 Elliptic curve is defined as

$$y^2 = x^3 + 7$$

Taking the derivate gives

$$2 y dy = 3 x^2 dx$$

For the slope, we rearrange

$$m = dy / dx = (3 x^2) / 2y$$

This equation is used when dealing with a line or a point on the elliptic curve. An Elliptic Curve have no concept of division, the 2y must pass through the modulo inverse function.

In Elliptic Curves, two points are added to give a third point which will be also be on the same curve. In normal math's, when two points are added on a curve, the resultant point may not lie on the curve.

First a line is drawn joining the two points. Then this line is extended so that it touches the Elliptic Curve. This new point will obviously be on the line and the Elliptic Curve, both. The condition is that it must satisfy both the line and elliptic curve equations.

$$y = m x + v \text{ for the line A}$$

$$y^2 = x^3 + b \text{ for the elliptic curve where b is 7}$$

From the line equation, the value of v can be obtained as

$$v = y - mx$$

As the Elliptic curve has y^2 on the left, we square the line equations. From line A

$$y^2 = (mx + v)^2$$

Let's expand both sides

$$y^2 = m^2x^2 + 2mvx + v^2$$

Since both the line and elliptic curve have the same left hand side, they can be equated. This is also because this point is present on the line and elliptic curve.

$$m^2x^2 + 2mvx + v^2 = x^3 + b$$

Everything is moved to the right-hand side

$$0 = x^3 - m^2x^2 - 2mvx + (b - v^2)$$

The above equation is a cubic polynomial in x because there is an x^3 . This is a very difficult polynomial to solve as there are exactly three solutions. We took someone's word on this, please take ours.

$$0 = (x-x_1) * (x - x_2) * (x - x_3)$$

The expanded version of the above equation is

$$x^3 - (x_1 + x_2 + x_3) x^2 + (x_1x_2 + x_2x_3 + x_1 x_3) x - x_1x_2x_3$$

Here x_1 and x_2 are the two points on the curve and x_3 is the new point.

As both the left-hand sides are 0, they can be equated.

So, simply take the x^2 coefficients to get

$$x_1 + x_2 + x_3 = m^2$$

Only x_3 , the new point is crucial here

$$x_3 = m^2 - x_1 - x_2$$

However, if it is Point doubling then there is a single point x_1 and the equation now becomes

$$x_3 = m^2 - 2x_1$$

The actual addition point is not this point but a flip or a mirror to this point. If you recall, y had to be changed to $-y$.

Now that we have a x , the y coordinate is on the line, so it becomes

$$y_3 = mx_3 + v$$

Flipping this point gives

$$y_3 = -(mx_3 + v)$$

Substituting v gives the following

$$y_3 = -(mx_3 + y_1 - mx_1)$$

Removing m which is common

$$y_3 = -(m * (x_3 - x_1) + y_1)$$

$$y_3 = m * (x_1 - x_3) - y_1$$

This knowledge will be applied to our own ECC match.

Calculating the Public Key Without Using Sage

```
ch2019.py
from bitcoin import *
Pcurve = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
N=0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
x = Gx
y = Gy
def modinv(a,n):
    lm, hm = 1, 0
    low, high = a % n, n
    while low > 1:
        ratio = high/low
        nm, new = hm - lm * ratio, high - low * ratio
        lm, low, hm, high = nm, new, lm, low
    return lm % n

m = ((3 * x * x) * modinv((2 * y),Pcurve))
print "Slope ", m
```

```
x3 = (m * m - 2 * x) % Pcurve
y3 = (m * (x - x3) - y) % Pcurve
pub = privtopub(0x2)
print "x " , x3
print "y " , y3
if (x3 == pub[0]) and (y3 == pub[1]):
    print "All is good"
else:
    print "Vijay Mukhi is an embecile"
```

Output

```
Slope
75662849025312301406793370858350329584492907588223948554043135653491003
36188305011441051952853644895621574418377968636140709566364984567929108
98817389940831543204657474297072356228690296487944931559885281889207062
770782744748470400
x
89565891926547004231252920425935692360644145829622209833684329913297188986597
y
12158399299693830322967808612713398636155367887041628176798871954788371653930
All is good
```

Let's now actually put our knowledge of modulo inverse, addition and point doubling into calculating the public key of a private key with a value of 2; without using Sage.

It starts with the same basic constants of the secp256k1 Elliptic Curve. We first use the modinv function to find the modular inverse of a number over a finite field defined by a prime number. Then, the slope of the Point m is determined using the same equation derived earlier.

As stated earlier, there is no division in the Elliptic Curves of Bitcoin. The modinv function is called with two parameters. The first parameter is $2 * y$ and the second parameter is good old Pcurve. The x and y variables defined here are the standard Generator point defined by the secp256k1 standard, Gx and Gy.

The slope of the curve is a very big number. Now, to find the x or x3 coordinate of this curve, there is Point doubling. And the equation is the slope squared minus $2 * x$ of the generator point. Everything outside of Sage asks for the modulus of the number, it cannot exceed Pcurve. The y or y3 coordinate is also calculated using an earlier formula.

The public key of private key 2 is determined using the bitcoin library. It matches the newly calculated x and y coordinates. The reason being, the public key of private key of 2 is determined by multiplying the private key 2 with the Generator point, point doubling does the same thing.

The Public Key of Private Key 2 Without Sage

```
ch2020.py
from bitcoin import *
Pcurve = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
N=0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
```



```

def modinv(a,n):
    lm, hm = 1 , 0
    low, high = a % n,n
    while low > 1:
        ratio = high/low
        nm, new = hm - lm * ratio, high - low * ratio
        lm, low, hm, high = nm, new, lm, low
    return lm % n

def ECadd(x1 , y1 ,x2 , y2):
    m = ( (y2 - y1 ) * modinv( x2 - x1 , Pcurve)) % Pcurve
    x3 = ( m * m - x1 - x2 ) % Pcurve
    y3 = ( m * ( x1 - x3 ) - y1 ) % Pcurve
    return (x3, y3)

def ECdouble(x1 , y1):
    m = ( (3 * x1 * x1 ) * modinv( 2 * y1 ), Pcurve)) % Pcurve
    x3 = (m * m - 2 * x1) % Pcurve
    y3 = (m * ( x1 - x3) - y1) % Pcurve
    return (x3,y3)

(xt,yt) = ECdouble( Gx , Gy )
(x,y) = ECadd( xt , yt, Gx , Gy)

pub = privtopub(3)
if (x == pub[0]) and (y == pub[1]):
    print "All is good"
else:
    print "Vijay Mukhi is an embecile"

```

Output

All is good.

One more step. Let's find out the public key of private key 3. The number 3 comes from first doubling number 1 by 2 or multiplying the point by 2. It is incremented by one so the value is 3.

Here we are not adding two points on a curve to get a third point. Instead, we are only drawing a line through these two points on the curve x_1, y_1 and x_2, y_2 and ensuring that it intersects the Elliptic Curve at a third point. This point is flipped to get the actual point. This is standard ECC math's explained on the Internet. Fortunately for us, we had many people on the internet who explained this better than us. Our only claim to fame, we simply broke up the code into smaller more manageable sections.

The program has a function called ECadd, a name copied from a YouTube video on Elliptic Curves. We give credit where its due. We could have passed this code as our own by simply changing function and variable names but we didn't.

The slope m is calculated using the equation $y_2 - y_1 / x_2 - x_1$. Plus, the $(x_2 - x_1)$ is replaced with a modular inverse over Pcurve. The x_3 value of the point is not like point doubling. There was only one point then, here there are two points, x_1 and x_2 . Hence, it is subtracted.

The y coordinate of the point is calculated in the same way. However, for the Point doubling function, only one set of coordinates are used. The same code is repeated wherein the Generator point is doubled to give the public key of 2. To add the new Point on the Elliptic Curve, the ECadd function is called and it generates the public key of private key 3.

In simple English, it is double + add or $1 * 2 + 1$. Finally, there is a check with the privtopub function to ensure that we have the right answer. Trust + verify is the maxim of life.

Calculating the public key of a private key 35

```
ch2021.py
from bitcoin import *
Pcurve = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
N=0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
privkey = 35
def modinv(a,n):
    lm, hm = 1, 0
    low, high = a % n, n
    while low > 1:
        ratio = high/low
        nm, new = hm - lm * ratio, high - low * ratio
        lm, low, hm, high = nm, new, lm, low
    return lm % n

def ECadd(x1, y1, x2, y2):
    m = ( (y2 - y1) * modinv( x2 - x1, Pcurve)) % Pcurve
    x3 = ( m * m - x1 - x2 ) % Pcurve
    y3 = ( m * ( x1 - x3 ) - y1 ) % Pcurve
    return (x3, y3)

def ECdouble(x1, y1):
    m = ( (3 * x1 * x1) * modinv( 2 * y1 ), Pcurve)) % Pcurve
    x3 = (m * m - 2 * x1) % Pcurve
    y3 = (m * ( x1 - x3 ) - y1) % Pcurve
    return (x3,y3)

def EccMultiply(Gx, Gy, pkey):
    print bin(pkey)
    keybin = bin(pkey)[2:]
    print keybin
    print len(keybin)
    x3, y3 = Gx, Gy
    for i in range(1, len(keybin)):
        print i, keybin[i]
        x3, y3 = ECdouble(x3, y3);
        if keybin[i] == "1":
            x3, y3 = ECadd(x3, y3, Gx, Gy);
    return (x3, y3)

(x, y) = EccMultiply(Gx, Gy, privkey)
pub = privtopub(privkey)
if (x == pub[0]) and (y == pub[1]):
```

```

    print "All is good"
else:
    print "Vijay Mukhi is an embecile"

```

Output

```

0b100011
100011
6
1 0
2 0
3 0
4 1
5 1
All is good.

```

So far, our private keys have been very small numbers. Let's now find the public key of a private key 35, which is stored in variable `privkey`. There is one more function created called `EccMultiply`, which is given three parameters, the Generator Point `G`'s `x` and `y` coordinates as well as the private key. The function returns the `x` and `y` coordinates of the public key. These point coordinates are checked with the return value of the original `privtopub` bitcoin module function.

There is a function in Python called `bin` that returns the binary representation of a number but with a prefix of `0b`.

The `bin` function returns a string and the slice operator is used to remove the first 2 characters. The length of this string is 6 in our case, as the private key is only 35. Then, the values in `x3` and `y3` are initialized to the Generator point's `x` and `y` values.

The for loop starts reading the string from offset 1 and not 0.

There is a certain logic here as the value of the point on the Elliptic Curve is first doubled. There are three consecutive 0 bits and it ends with two 1 bits. To start with, we are at the generator point.

The function `ECdouble` doubles the value of the point. Then there is a check for the current binary digit to be 1. If so, the Generator point is added to the current point. The current point position on the Elliptic curve is stored in `x3` and `y3`. The first bit is ignored where the variable `i` has a value of 0. The return values of `x3` and `y3` are checked with the public key `x` and `y`, values returned by the `privtopub` function. The length of the returned `bin` string decides on the loop iterations.

In the end, we have successfully computed the public key of a large enough private key. And proved that doubling and adding is better than multiplication, we can live without multiplication.

Calculating the Signature Without Using Sage

```

ch2022.py
from bitcoin import *
Pcurve = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8

dA = 3
k = 4
e = 1

```

```
def modinv(a,n):
    lm, hm = 1 , 0
    low, high = a % n,n
    while low > 1:
        ratio = high/low
        nm, new = hm - lm * ratio, high - low * ratio
        lm, low, hm, high = nm, new, lm, low
    return lm % n

def ECadd(x1 , y1 ,x2 , y2):
    m = ( (y2 - y1 ) * modinv( x2 - x1 , Pcurve)) % Pcurve
    x3 = ( m * m - x1 - x2 ) % Pcurve
    y3 = ( m * ( x1 - x3 ) - y1 ) % Pcurve
    return (x3, y3)

def ECdouble(x1 , y1):
    m = ( ( 3 * x1 * x1 ) * modinv( ( 2 * y1 ), Pcurve)) % Pcurve
    x3 = (m * m - 2 * x1) % Pcurve
    y3 = (m * ( x1 - x3) - y1) % Pcurve
    return (x3,y3)

def EccMultiply(Gx , Gy , pkey):
    keybin = bin(pkey)[2:]
    x3 , y3 = Gx , Gy
    for i in range (1, len(keybin)):
        x3 , y3 = ECdouble(x3 , y3);
        if keybin[i] == "1":
            x3 , y3 = ECadd(x3 , y3 , Gx , Gy);
    return (x3 , y3)

QAx, QAy = EccMultiply(Gx , Gy , dA)
xr, yr = EccMultiply( Gx , Gy , k)
r = xr % n
s = ((e + r * dA ) * (modinv(k , n ) ) ) % n

w = modinv( s , n )
xu1, yu1 = EccMultiply(Gx , Gy, (e * w) % n )
xu2, yu2 = EccMultiply(QAx ,QAy ,( r * w ) % n )
x , y = ECadd( xu1 , yu1 , xu2 ,yu2 )
print r == x
```

Output

True

Two programs are merged to square the circle. First the public key is computed from a private key dA. Aka last program.

Then the message e is signed using the same code written in the last chapter. Further, it is verified that the private key signed the message. The program may look different because of change in some variable names.


```
r2 = xr % n
s2 = ((e2 + r2 * dA)*(modinv(k,n))) % n
k1 = (e1 - e2)
k2 = (s1 - s2)
k2 = modinv(k2 , n)
k0 = (k1 * k2) % n
if k0 == k:
    print "All is good for Random Number %d" % k
else:
    print k0 , k
    exit()
k = k + 1
```

Output

```
All is good for Random Number 1
All is good for Random Number 2
All is good for Random Number 3
```

There must be some way to stress-test our user-written code. Earlier for signing, we had two programs that figured out the random number and the private key used from the same random number.

The same code is reused but it is placed in a for loop to generate a different random number k each time.

Stress Testing With the Random Number and Private Key

```
ch2024.py
from bitcoin import *
Pcurve = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
Gx = 0x79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8

def modinv(a,n):
    lm, hm = 1 , 0
    low, high = a % n,n
    while low > 1:
        ratio = high/low
        nm, new = hm - lm * ratio, high - low * ratio
        lm, low, hm, high = nm, new, lm, low
    return lm % n

def ECadd(x1 , y1 ,x2 , y2):
    m = ( (y2 - y1 ) * modinv( x2 - x1 , Pcurve)) % Pcurve
    x3 = ( m * m - x1 - x2 ) % Pcurve
    y3 = ( m * ( x1 - x3 ) - y1 ) % Pcurve
    return (x3, y3)

def ECdouble(x1 , y1):
    m = ( ( 3 * x1 * x1 ) * modinv( ( 2 * y1 ), Pcurve)) % Pcurve
```

Output

391

In this example, there are two for loops, one to change the random number k and the other to change the private key dA . All code of the previous programs is used here.

Our long journey with cryptography comes to an end. Now, to send a transaction to a miner with all the knowledge gained so far.

Once again, if we had to rewrite parts of the book. We are trying very hard to do away with this mind block when understanding the mathematics behind cryptography. Words fail us and not code, when it comes to explaining basic mathematics concepts. Not our finest hour!!!

CHAPTER 21

Sending Our Own Transaction

This chapter provides some more insight into the knowledge gained so far. We try to clear all ambiguity and highlight some more features. Our goal as stated all the time, is to simply send a transaction to the Bitcoin ecosystem using our own code. Ok, kind of written by us.

This means it will cover the whole cycle, the output that supplies the Bitcoins, to signing the transaction, and then sending the transaction to a miner. What is required is a private key created by the Bitcoin-cli client, a public key and a Bitcoin address. Also, please send some Bitcoins to your Bitcoin address and fill up your wallet.

Finally, in the programs wherever there are hardcoded keys, please replace them with your set of values. To identify these variables, we have a naming convention. The word is hard, this is a short form for hard coded. So, in the program, every time there is a variable name starting with hard, replace the value with your values. For example, the Bitcoin address in all the programs was saved in a variable named baddr, in this chapter it is called hardbaddr.

Most of the code is based on the functions in Python libraries like pybitcointools and bitcoin. We have simply removed the error checks and other unnecessary complexities.

The first program figures out which outputs are owned by our Bitcoin address and not used so far.

Fetching Bitcoin Address Usage Using Blockchain.info

```
ch2101.py
import bitcoin
import json
hardbaddr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
url = 'https://blockchain.info/address/%s?format=json' % (hardbaddr)
print url
data = bitcoin.make_request(url)
print data[:300]
jsonobj = json.loads(data)
print type(jsonobj)
print "Total Sent %s" % jsonobj['total_sent']
print "Total recieved %s" % jsonobj['total_received']
txs = []
txs.extend(jsonobj["txs"])
print "No of transaction %s" % len(txs)
```

Output

```
https://blockchain.info/address/1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n?format=json
{
```

```
    "hash160": "d7ce0c2c237ec0ec04931335377a87d16b9865ad",
    "address": "1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n",
    "n_tx": 57,
    "total_received": 2052600,
    "total_sent": 1602000,
    "final_balance": 450600,
    "txs": [
    {
        "lock_time": 0,
        "result": 0,
        "ver": 1,
        "size": 191,
        "inputs": [
```

```
<type 'dict'>
Total Sent 1602000
Total recieved 2052600
No of transaction 50
```

The output displayed in the browser of the website, blockchain.info can be achieved using any programming language, Python included. Enter the above URL in any browser and it will give the same output as shown by this program.

The Bitcoin address in the program starting with 1Lg is a valid Bitcoin address, so it will work in the same way in any application. All our transactions are stored in the blockchain which is for public consumption. Your Bitcoin address will work as well. However, programs using just our Bitcoin addresses cannot be used to transfer Bitcoins as who will pay the miners. Our Bitcoins addresses have no money left in them.

Now to the mechanics of the web url. Every transaction carries a certain address, so the word 'address' is placed in the URL, followed by the actual Bitcoin address. The `hardbaddr` variable fills in the Bitcoin address dynamically. Our URL also works as advertised.

The words json stands for the JavaScript Object Notation; Python also loves it. It is one of the most widely used formats in the world today. Without the format parameter, the output gets shown in an HTML page in a browser.

The Bitcoin library function `make_request` executes this url. Very soon, we will replace this function with our code. The output received from this request is in json. This output with its field names is unique to the blockchain api only. Any other web service will not work with this code.

The key `n_tx` has the number of transactions for our Bitcoin address starting with 1Lg. In our case, it is only 57. Your mileage will vary.

The next three keys refer to the following: number of Bitcoins received from everyone, the Bitcoins whose ownership is transferred and finally the remaining Bitcoin balance. These values can be confirmed with our Bitcoin address in blockchain.info. The first 300 bytes of the raw json file is then displayed.

As far as possible, code from the module `pybitcointools` or the module `bitcoin` is kept intact, nothing has been changed, including the variable names.

Let's use a Python dictionary now. The json module is imported in the program. There is a function called `loads` that takes a json format string and converts it into a Python dictionary. It becomes easy with dictionaries especially when displaying the required fields as the keys associated with the values are available. To assert this, the keys `total_sent` and `total_received` are displayed on the screen. The variable `jsonobj` is of type Dictionary.

The most widely used key returned by the blockchain web services is called txs. This key is read into a list also called txs. Now, the extend function present in any list object is used to access the list stored in the key jsonobj["txs"]. The length of the txs list is only 50, whereas the n_tx key says 57 transactions. One of the reasons for having more than 50 transactions associated with our Bitcoin address. A rule set by the blockchain explorer simply for optimization, so that the blockchain server is not burdened with too much work.

When you run the same code, the number of transactions would be in the 90s.

Retrieving all the Bitcoin Address Transactions, Just Not the First 50

```
ch2102.py
import bitcoin
import json
hardbaddr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
txs = []
offset = 0
while 1:
    url = 'https://blockchain.info/address/%s?format=json&offset=%s' % (hardbaddr, offset)
    print url
    data = bitcoin.make_request(url)
    jsonobj = json.loads(data)
    txs.extend(jsonobj["txs"])
    if len(jsonobj["txs"]) < 50:
        break
    offset += 50
print "No of transaction %s" % len(txs)
```

Output

```
https://blockchain.info/address/1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n?format=json&offset=0
https://blockchain.info/address/1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n?format=json&offset=50
No of transaction 57
```

The program reads the 57 transactions associated with our Bitcoin address in a list called txs. The web url has a parameter called offset that specifies the starting point of the transaction number. No ending point is defined as the default limit of the blockchain api is 50. The offset variable is initialized to 0.

As the value of the variable offset is 0, the first 50 rows are retrieved and then the variable offset is increased by 50 at the end of the loop. In the next round, rows or transactions from 51 to 100 are picked up. Then 101 to 150 and so on and so forth. The loop has no condition as it is not clear as to how many transactions are associated with a single Bitcoin address. This while loop will never stop.

A check is performed if the number of transactions returned in the loop is less than 50. If this condition is true it indicates the end of the number of rows. The loop is terminated with the break statement. In our case, the balance 7 transactions are returned in the second iteration of the loop itself.

The data sent by the remote server will be different each time. So, these rows are concatenated to the txt list using the same extend function call.

To re-check if all the transactions in our list are obtained, the size is displayed which is the magic number 57 and not 50.

Rewriting the history function our way

```
ch2103.py
import bitcoin
import json
hardbaddr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
txs = []
offset = 0
while 1:
    url = 'https://blockchain.info/address/%s?format=json&offset=%s' % (hardbaddr, offset)
    data = bitcoin.make_request(url)
    jsonobj = json.loads(data)
    txs.extend(jsonobj["txs"])
    if len(jsonobj["txs"]) < 50:
        break
    offset += 50
print "Original Number of Transactions %d" % len(txs)
outs = {}
cnttxs = 0
cntout = 0
cntbaddr = 0
for tx in txs:
    cnttxs = cnttxs + 1
    for o in tx["out"]:
        cntout = cntout + 1
        if o.get('addr') == hardbaddr:
            cntbaddr = cntbaddr + 1
        key = str(tx["tx_index"]) + ':' + str(o["n"])
        outs[key] = {
            "address": o["addr"],
            "value": o["value"],
            "output": tx["hash"] + ':' + str(o["n"]),
            "block_height": tx.get("block_height", None)
        }
    if cntbaddr == 2:
        print "cnttxs %d cntout %d cntbaddr %d" % (cnttxs, cntout, cntbaddr)
        print o
        print "tx_index=%s:n=%s" % (str(tx["tx_index"]), str(o["n"]))
        print "key=%s" % key
        print "block_height=%s" % tx.get("block_height", None)
        print "Transaction Hash is %s" % tx["hash"]
        print outs

print "Number of Transactions with our Bitcoin address %d" % len(outs)
print cnttxs, cntout, cntbaddr
```

Output

Original Number of Transactions 57

```

cnttxs 8 cntout 12 cntbaddr 2
{'u'addr': u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n', u'script':
u'76a914d7ce0c2c237ec0ec04931335377a87d16b9865ad88ac', u'spent': True,
u'value': 56000, u'n': 3, u'tx_index': 204854525, u'type': 0}
tx_index=204854525:n=3
key=204854525:3
block_height=446079
Transaction Hash is
222a037f5106a8abfe87f29a556c24b78bf2bf11283aa053800c087774e438c4
{'204854525:3': {'output':
u'222a037f5106a8abfe87f29a556c24b78bf2bf11283aa053800c087774e438c4:3',
'block_height': 446079, 'value': 56000, 'address':
u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}, '206741458:0': {'output':
u'04ffb821f8cef805d104615333c6e6758601fdeb1a6f48fd6621595915cbe38c:0',
'block_height': None, 'value': 55000, 'address':
u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}}
Number of Transactions with our Bitcoin address 32
57 110 32

```

The program comes closer to displaying an output similar to the one by the history function. The list called txs has 57 members/transactions associated with our Bitcoin address starting with 1Lg. Using the for loop construct, each transaction tx in the txs list is analyzed. One of the reasons for converting a json object to a Python list. The variable cnttxs is incremented once at the start of the for loop. Since there are 57 transactions, the last print will display the value of cnttxs as 57. This variable ensures that every transaction is scanned.

A Bitcoin transaction has multiple inputs and outputs. The task is to run through each output to check if the Bitcoin address in that output equals our Bitcoin address starting with 1Lg. The out key in the transaction tx contains all the outputs only and not the inputs.

The for loop is the inner loop and it scans through all the outputs found in a single transaction. Here, the variable cntout is increased by 1. This variable counts the number of outputs in all the transactions. The total number of outputs in the 57 transactions are 110. A rough average of 2 outputs per transaction. A transaction can also have only 1 output.

The get method of the list is used in place of the key. The reason being, the get method knows how to handle exceptions when a key is missing. The if statement being true indicates that the output matches our Bitcoin address. The key addr in the output has our Bitcoin address.

In the if statement, a variable named key is created and initialized to the value stored in key, tx_index. Its current value is 204854525, which is a very random looking number. The unique value in key tx_index is associated with a transaction. Please confirm the value of this key with the output we have displayed. The output is displayed only when the variable cntbaddr has a value of 2 and not 1. We do not want too many outputs shown.

The key n has a value of 3. The reason being the output that contains our Bitcoin address, is the fourth in the list of Bitcoin outputs.

If you enter the transaction hash starting with 222 in your explorer, the fourth output displayed is a Bitcoin address starting with 1Lg. This transaction hash value is the value of the hash key.

A key must be a unique value. To achieve this uniqueness, the value in key tx_index is concatenated with the output index represented by n, but separated by a .:

Moving on, one item is added to the dictionary outs where this unique key is used as the key and its value is a dictionary of four values. The key names are chosen by us and they are as follows:

The addr key has the Bitcoin address.

The value key is for the Bitcoins value or amount.

The most important key is the output key which is the hash value of the transaction that contains the output along with the output index. The original code separated these values by a :, so we follow it blindly.

Finally comes the block height.

An old rule. Unless a Bitcoin address is present in an earlier output it cannot be used in another input later. This does not imply that the output cannot be spent as it may be spent already.

The dictionary o states that the output is spent because the key spent has a value of true.

The total number of outputs found so far, are 12 and we are on transaction number 8 only when the variable cntbaddr has a value of 2. A key is added to the outs dictionary in an if statement, thus adding only one output. This is a kludge, unique for this case only.

Exercise. Please write the number 1 which stands for block number in blockchain.info. Then confirm that the only transaction hash value displayed is

```
0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098.
```

At the bottom, there is a Bitcoin address which goes as

```
12c6DSiU4Rq3P4ZxziKxziL5LmMBrzjrJX.
```

Any blockchain explorer is smart enough to know from the size of data whether it is a block number or Bitcoin address or a hash value. It also uses some intelligence to figure out whether it is a block hash value or a transaction hash value.

We then write the following URL in any browser.

```
https://blockchain.info/block/00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048?format=json
```

This URL is obtained by placing the Bitcoin address found in the first block.

The blockchain explorer has limited choice but to display the json representation of block 1. The data entered is the hash value of block 1. Alternatively, the URL could be formed with the value 1 and then concatenated with ?format=json. When this URL is typed in the explorer search window, the output received is displayed below:

```
{
  "hash": "00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048",
  "ver": 1,
  "prev_block": "00000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
  "mrkl_root": "0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098",
  "time": 1231469665,
  "bits": 486604799,
  "fee": 0,
  "nonce": 2573394689,
  "n_tx": 1,
  "size": 215,
  "block_index": 14850,
  "main_chain": true,
```

```

    "height":1,
    "tx":[
    {
        "lock_time":0,
        "ver":1,
        "size":134,
        "inputs":[
            {
                "sequence":4294967295,
                "script":"04ffff001d0104"
            }
        ],
        "time":1231469665,
        "tx_index":14854,
        "vin_sz":1,
        "hash":"0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098",
        "vout_sz":1,
        "relayed_by":"0.0.0.0",
        "out":[
            {
                "spent":false,
                "tx_index":14854,
                "type":0,
                "addr":"12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX",
                "value":5000000000,
                "n":0, "script":"410496b538e853519c726a2c91e61ec11600ae1390813a6
27c66fb8be7947be63c52da7589379515d4e0a604f814
1781e62294721166bf621e73a82cbf2342c858eeac"
            }
        ]
    }
  ]
}

```

The transaction index key is 14584 and not 2 as this is the second Bitcoin transaction in the history of the Bitcoin universe. The tx_index key is a unique number. It does not denote in any way the order of the transactions.

One Last Stab at the History Function

```

ch2104.py
import bitcoin
import json
hardbaddr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
txs = []
offset = 0
while 1:
    url = 'https://blockchain.info/address/%s?format=json&offset=%s' % (hardbaddr, offset)
    data = bitcoin.make_request(url)

```

```
    jsonobj = json.loads(data)
    txs.extend(jsonobj["txs"])
    if len(jsonobj["txs"]) < 50:
        break
    offset += 50
print "Original Number of Transactions %d" % len(txs)
outs = {}
cnttxs = 0
cntout = 0
cntbaddr = 0
for tx in txs:
    cnttxs = cnttxs + 1
    for o in tx["out"]:
        cntout = cntout + 1
        if o.get('addr') == hardbaddr:
            cntbaddr = cntbaddr + 1
        key = str(tx["tx_index"])+':'+str(o["n"])
        outs[key] = {
            "address": o["addr"],
            "value": o["value"],
            "output": tx["hash"]+':'+str(o["n"]),
            "block_height": tx.get("block_height", None)
        }
print "Number of Transactions with our Bitcoin address %d" % len(outs)
print cnttxs , cntout , cntbaddr
cnttxs = 0
cntinputs = 0
cntprev = 0
cntbaddr = 0
cnt = 0
for tx in txs:
    cnttxs = cnttxs + 1
    for i, inp in enumerate(tx["inputs"]):
        cntinputs = cntinputs + 1
        if cntinputs == 1:
            print inp
            print i
            if "prev_out" in inp:
                cntprev = cntprev + 1
                if cntprev == 1:
                    print inp["prev_out"]
                    if inp["prev_out"].get("addr") == hardbaddr:
                        cntbaddr = cntbaddr + 1
                    key = str(inp["prev_out"]["tx_index"]) + ':' + str(inp["prev_out"]["n"])
                    if cntbaddr == 1:
                        print key
                        if outs.get(key):
```



```

    cnt = cnt + 1
    outs[key]["spend"] = tx["hash"] + ':' + str(i)
    if cnt == 1:
        print outs[key]
        print outs[key]["spend"]
    print cnttxs, cntinputs, cntprev, cntbaddr, cnt
    ans = [outs[k] for k in outs]
    print "Final Number of transactions %d" % len(ans)
    print ans[0]
    a = ["a1", "a2", "a3"]
    for i, ans in enumerate(a):
        print i, ans

```

Output

Original Number of Transactions 57

Number of Transactions with our Bitcoin address 32

57 110 32

```

inp==== {u'script':
u'473044022068d45bb4fa22b3cc2b10281c8e3cdda41be02932c6105c43f1075685f7938
9e00220699bf5a6a1227ab1264a2633bbae1af63a9082bdbfe5449add86c5ffbf77fb4301
21038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3',
u'prev_out': {u'addr': u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n', u'script':
u'76a914d7ce0c2c237ec0ec04931335377a87d16b9865ad88ac', u'spent': True,
u'value': 56000, u'n': 3, u'tx_index': 204854525, u'type': 0}, u'sequence': 4294967295}
0
prev_out--- {u'addr': u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n', u'script':
u'76a914d7ce0c2c237ec0ec04931335377a87d16b9865ad88ac', u'spent': True,
u'value': 56000, u'n': 3, u'tx_index': 204854525, u'type': 0}
key^^^^^ 204854525:3
outs%%% {u'output':
u'222a037f5106a8abfe87f29a556c24b78bf2bf11283aa053800c087774e438c4:3',
'block_height': 446079, 'spend':
u'04ffb821f8cef805d104615333c6e6758601fdeb1a6f48fd6621595915cbe38c:0',
'value': 56000, 'address': u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}
spend&&&&
04ffb821f8cef805d104615333c6e6758601fdeb1a6f48fd6621595915cbe38c:0
57 57 57 26 26
Final Number of transactions 32
{u'output':
u'2798f73adddd8d4b407ed6153045bdf6c5d4d72b6d1454d20456cd5eece4447e:3',
'block_height': 446079, 'spend':
u'498a8a4ca188c655a9c0d43e6cb43d99501d2874a190feb926e37d3255d2d8d7:0',
'value': 58900, 'address': u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}
0 a1
1 a2
2 a3

```

The spend key in the program determines if the output is spent or not. It specifies the transaction and the index that spent the output. If the output has not been spent, then the key spent is absent. The history function does it differently. The history function reveals more information than required and hence it is more difficult to understand. The original history function can handle multiple addresses, ours has only one. This makes the program simpler to understand.

The variable tx is a dictionary representing each transaction, and once again every transaction in the txs list is scanned. The variable cnttxs as before, counts the number of transactions present with our Bitcoin address, a measly 57. The enumerate function towards the end of code is like a list. It gives a unique number to every list value, starting from 0, thus incorporating a free index number into the list. Unfortunately, in our case, the variable i always has a value of 0 as the inputs have only one row/record. The variable cntinputs has a value of 57 as every transaction has only one input. It confirms we are on the right track.

The input displayed has a key called prev_out which discloses the output plus the index of the Bitcoin supplied and the spend key. The other keys in the dictionary key prev_out are irrelevant for now. The variable cntprev also has a value of 57 as every input has the prev_out key.

The if statement is an additional error check that can be ignored as all inputs contain this key. The next if statement checks if the addr key is our Bitcoin address.

The program output reveals that out of the 57 transactions, only 26 of them have our Bitcoin address. This implies that 31 transactions do not carry our Bitcoin address but are included in the transaction list.

Why?

A Bitcoin address is present either in the output or in an input that refers to an output (using a transaction hash value). The earlier code also made the same conclusion.

The if statement results in true when the addr key matches our Bitcoin address. As before, the tx_index and the output index key n is our key and it is called key. The key n is part of the key called prev_out. The value displayed is 204854525:3.

The length of outs is 26. This is the value of the variable cnt. A check is performed on the key we just calculated to be present in the outs dictionary as a key. This check is not needed, though. Same keys are added to the outs dictionary as well. Hence, the outs.get(key) will always return true. In a sense, the if statement is redundant. Therefore, it is a good idea to read the code some multiple times. This happens as we write code in the wee hours of the morning and then realize our mistake. Sloppy programming is another way of describing such code.

Then a key called spend is created. The value in the spend key is the transaction hash value and the row index number or the output index. The output displayed of the transaction is our Bitcoin address. Either str(i) or the n key can be used for this purpose.

The keys and the dictionary are insignificant for now, so using one line of Python i.e. a for statement, the keys are removed and the dictionary values are put into a simple list. This list is called ans. This list has a length of 32.

Let's assume that there are 32 outputs that carry our Bitcoin address. These outputs are the ones that have the spent key as true. What is required now is to remove the outputs where an input has a transaction hash value and the output index refers to that output. The outputs that have been spent before are removed to give a total count of available unspent outputs.

So, in all there are a total of 32 transactions. As mentioned earlier, the length of the ans list is 32. The variable cntprev gives a count on the number of inputs referring to the outputs in the past. The value is 26. The only outputs that have no inputs refer to them are 32 - 26 or 6.

We will explain this in detail later.

Talking to the Blockchain Explorer Using Standard Python Code

```
ch2105.py
import urllib2
hardbaddr = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
url = 'https://blockchain.info/address/%s?format=json' % (hardbaddr)
urlheaders = {
    "User-agent": "Vijay Mukhi",
}
req = urllib2.Request(url, data=None, headers=urlheaders)
print req.get_full_url()
print req.headers
print req.get_method()
data = urllib2.urlopen(req).read()
print data[115:126]
```

Output

```
https://blockchain.info/address/1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n?format=json
{'User-agent': 'Vijay Mukhi'}
GET
    "n_tx":57,
```

The blockchain.info api can return a million different things. Seriously, it is a very complex API. Using the standard Python library, a connection is established to the blockchain.info http server and a simple GET request is sent. With the http protocol, there is either a get or a post method as a request. In a get method, the data travels along with the URL and in a post, the data is sent as a separate packet.

A dictionary called urlheaders is created with only one key-value pair. The key name is set to User-agent and the value has my name. Since this header is optional, blockchain.info does not complain about the value of this field. As we mentioned earlier in the networking chapter, this header is optional.

The library urllib2 does all the work, we simply give it our url, the data and the optional headers fields. The function urlopen sends the http request over and the read function waits for a response. A request object is returned and we print the url, method and headers using standard functions in this object. Internally, they all call the sockets send and recv functions with error checks.

We display a very small part of the output, a number we are all familiar with now, the number of transactions 57.

In the future, a library like urllib2 will be used very often to talk to our blockchain explorer.

Private Keys Revisited

```
ch2106.py
import hashlib
def decode58(string):
    code_string = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
    val = 0
    while len(string) > 0:
        val = val * 58
        val = val + code_string.find(string[0])
```

```
    string = string[1:]
    return val
def encode58(val):
    code_string = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
    string = ""
    while val > 0:
        string = code_string[val % 58] + string
        #val //= 58
        val = val // 58
    return string
priv1 = '5HueCGU8rMjxEXxiPuD5BDku4MkFqeZyd4dZ1jvhTVqvbTLvyTJ'
print "priv1=%s" % priv1
base58 = decode58(priv1)
print "base59=%x" % base58
base59 = "%x" % base58
print "base59=%s" % base59
privkey = base59[2:-8]
print "privkey= %s" % privkey
print
privkey = '80' + privkey
print "priv=%s" % privkey
sha = hashlib.sha256(privkey.decode('hex')).digest()
print "sha =%s" % sha.encode('hex')
sha1 = hashlib.sha256(sha).digest()
print "sha1=%s" % sha1.encode('hex')
checksum = sha1[:4]
print "csum=%s" % checksum.encode('hex')
final = privkey + checksum.encode('hex')
print "last=%s" % final
zzz = int(final,16)
print "zzz =%x" % zzz
ans = encode58(zzz)
print "priv1=%s" % ans
if ans == priv1:
    print "All is well"
else:
    print "Vijay Mukhi is a embecile"
```

Output

```
priv1=5HueCGU8rMjxEXxiPuD5BDku4MkFqeZyd4dZ1jvhTVqvbTLvyTJ
base59=800c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1
d507a5b8d
base50=800c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1
d507a5b8d
privkey= 0c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1d
priv=800c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1d
```

```

sha =8147786c4d15106333bf278d71dadaf1079ef2d2440a4dde37d747ded5403592
sha1=507a5b8dfed0fc6fe8801743720cedec06aa5c6fca72b07c49964492fb98a714
csum=507a5b8d
last=800c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1d507a5b8d
zzz =800c28fca386c7a227600b2fe50b7cae11ec86d3bf1fbe471be89827e19d72aa1d507a5b8d
priv1=5HueCGU8rMjxEXxiPuD5BDku4MkFqeZyd4dZ1jvhTVqvbTLvyTJ
All is well.

```

While understanding the Bitcoin address, we had stopped short of explaining how a public key is obtained given a private key. With the public key, the Bitcoin address was computed. Then we learnt that a public key is one more point on an Elliptic Curve.

The private key comes in multiple formats in the Bitcoin world. When it is specified as a simple number like 420, the format is called decimal. When a wallet generates a private key, it uses a format called wif or the Wallet Interchange Format. This private key is much smaller than 32 bytes. If the private key starts with a 5, then the size is 51 hex digits, much smaller than the 64 digits we used. Therefore, this private key format is termed as wif_compressed. Smaller the size of the key, easier it is to handle. The smaller size also helps tag checksums at the end. The public key also come in various sizes.

The Bitcoin wiki has an example on how to convert a wif_compressed private key to an actual number and vice versa. However, there is no Python code or any code along with the example. We use the same private key the Bitcoin wiki uses but with Python code. Why reinvent the wheel when someone by far smarter has already done so?. It also confirms that we have done is correct.

In the program, the variable priv1 contains the wif_compressed private key which has a value 5HueCGU8rMjxEXxiPuD5BDku4MkFqeZyd4dZ1jvhTVqvbTLvyTJ. Let's convert this into a 32-byte number that created this wif_compressed private key in the first place.

The functions, decode and encode in the program have no relevance to the string class encode and decode functions. The reason being that the encode and decode methods take a string and return a string. Here, we have a base58 encoded string that represents our private key.

Let's now convert this base58 encoded string into an actual number. Converting a string into a number is called decoding. The decode58 function takes a base58 encoded string and returns a normal number as the return value. It takes a private key, which is stored in variable priv1 and returns a number which is saved in the variable base58.

Most of the code is repeated, not once but multiple times. The code_string variable stores the valid base58 characters. The val variable will be the final number giving this private key. After picking up one byte each time, the size of the string is reduced by 1. The while loop ends when the size of the string is 0.

The running total val is multiplied by 58 at the start of the loop. Then, the offset of the byte is determined in the code_string variable. This index offset is added back to the running total variable val. This variable val is the original number that created the private key in the first place.

The private key is displayed as a number in variable base58, as hex number. For reasons not known, the variable name is base59 and not base58. This number is converted into a string by simply using the "%x" print modifier as before. On the face of it, the string and number types look the same. Also ignore the print statements that have the wrong base variable names.

The privkey variable is the actual private key but it has the first byte 80 and the last 4-bytes (8 digits) for the checksum, conforming to the norms of a private address. The splice operator is used to remove the last 4 bytes. The -8 means ignore either last 8 digits or 4 bytes. Since it is an encoded string; the count is doubled. The first 2 is to ignore the 0x80. These numbers line up visually in a DOS box or terminal window. The variable privkey now contains 5 bytes less.

Now let's reverse the entire process to check our doings. Let's take this variable `privkey` which carries the decimal private key and get back the original `wif_compressed` private key.

First, 80 is added to the decimal private key value. This is very simple as the variable `privkey` is a simple string. Then, as always, the sha256 hash is calculated twice. The first 4 bytes of the second sha hash value is used as the checksum of the private key. This is stored in the checksum variable. This checksum is added to the very end of the private key after encoding the value. The private key is displayed after adding this value.

The checksum is a decoded string so it must be encoded while concatenating the strings. The variable `final` is a string and not a number. So, it is converted to a number using the `int` function. Finally, the values are the same, variable `zzz` is an integer, variable `final` is string. Mismatch of labels.

Now to the reverse of a decode. The number must be returned as a string. The `encode58` function takes a 256-base number and converts it into a base58 string. The string is returned. In the function, we use the same `code_string` variable, but the return value is a string called `string`. This variable `string` is initialized to an empty string at the start of the function.

Earlier the loop ended when the string length was 0, now we loop until the `val` variable or the number passed is positive. The remainder of `val` modded by 58 is taken each time in the loop. This gives the base58 character which is added at the beginning of the string.

The value of `val` is reduced using the operator `//`. This is technical, it is not division, but floor division. The string returned is the base58 encoded string of a normal or decimal number. Our take is that it makes no sense to have different formats for private keys to save some bytes.

The Bitcoin wiki has examples that can be converted to code, please use any programming language for the same.

Converting a wif Private Key to a Public Key

```
ch2107.py
import bitcoin
N =
11579208923731619542357098500868790785283756427907490438260516314151816
1494337
Gx =
55066263022277343669578718895168534326250603453777594175500187360389116
729240
Gy =
32670510020758816978083085130507043184471273380659243275938904335757337
482424
G = (Gx, Gy)
def decode58(string):
    code_string = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
    val = 0
    while len(string) > 0:
        val = val * 58
        val = val + code_string.find(string[0])
        string = string[1:]
    return val
priv = "KyH6Mm8sYmnT9cm2jMmxpae7CAvbCmmxQ1bNYsjAeMeeT2XD6KWp"
```

```

zz = decode58(priv)
xx = "%x" % zz
print "%x" % zz
print xx
print type(zz) , type(xx)
data = xx.decode('hex')
print len(data)
print data.encode('hex')
data = data[1:-4]
print data.encode('hex')
print len(data)
data = data[:32]
print data.encode('hex')
print len(data)
privkey = int(data.encode('hex'), 16)
if privkey >= N:
    raise Exception("Invalid privkey")
multiply = bitcoin.fast_multiply(G, privkey)
byte = str(2+(multiply[1] % 2))
print "First Byte %s" % byte
bb = "%x" % multiply[0]
pub = '0' + byte + bb
if pub == "028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b":
    print "All is good"
else:
    print "Vijay Mukhi is an embecile"

```

Output

```

803d6a2f408fe58dabce126718a06a655a4b49625572ab2eb1e9b6e094f11e1832013b0b0
1d1
803d6a2f408fe58dabce126718a06a655a4b49625572ab2eb1e9b6e094f11e1832013b0b0
1d1
<type 'long'> <type 'str'>
38
803d6a2f408fe58dabce126718a06a655a4b49625572ab2eb1e9b6e094f11e1832013b0b0
1d1
3d6a2f408fe58dabce126718a06a655a4b49625572ab2eb1e9b6e094f11e183201
==== 33
3d6a2f408fe58dabce126718a06a655a4b49625572ab2eb1e9b6e094f11e1832
- 32
First Byte 2
All is good

```

This program converts a wif_compressed private key to a public key. Only one function from a function library is used, the rest of the code is written by us.

Since the private key is in the wif_compressed format, our good old decode58 function is used to convert it into a normal

base 256 number. As a thumb rule, whenever there is a base58 encoded string, use function decode58 to give a number. For a normal number which is a base256 number, function encode58 converts it to a base58 encoded string.

The variable zz is a number and the variable xx with the same value is a string. Both variables look the same but they are of different data types. The string xx is decoded as it will be encoded later anyways for display purposes. We use decoded but display encoded, like we said multiple times in the past.

The length of the decoded data in zz is 38 bytes. As per the wif standard, the first byte is 80 and the last 4 bytes are for the checksum. The string is 33 bytes in size without these 5 bytes.

There are different private key formats, similarly there are different types of public key formats. A public key remains a public key, but its size changes. If the private key ends with a 01 after removing the checksum, the public key is a compressed public key.

Our private key ends with a 01 but in future, it may or may not. The slice operator [:32] picks up the first 32 bytes of the string. So, either way, the 01 is removed from the end of the string.

To summarize, from the 38 bytes, the first byte 0x80, checksum 4 bytes and maybe a 01 are sliced out. After removing these 6 bytes, the string becomes smaller with 32 bytes. The string must be converted to a number so, it is decoded using the int function.

Let's bring in an error check. The private key must not be larger than the Elliptic Curve Constant N. Makes sense. No-one can be above N.

As learnt earlier, a public key is a point of the secp256k1 Elliptic Curve. Now, the generator point G which is also a point on the Elliptic Curve is multiplied with the private key. This gives a new point on the Elliptic Curve.

This point is however special, it is called the public key. The Bitcoin module function fast_multiply determines if this point is on the Elliptic Curve. Once again, the public key is a tuple, that's because a point is two values, x and y on any curve. This has been explained earlier in the previous chapters.

One analogy we came across, spoke of an Elliptic Curve being shaped like a U. The public key can be on the left or the right side of the U. So, there are two equally valid values. Also, a square root can have two roots, plus and minus.

A public key is normally 65 bytes large, starting with a 04. It is followed by 32 bytes of the x coordinates and then 32 bytes of the y coordinates. On the other hand, a compressed public key is only 33 bytes large. The y coordinate is dropped so what's left, is the x. According to the official Bitcoin tutorial, the first byte of the compressed key can either be 02 or 03. If it is greater than the midpoint of the curve, the first byte is 03 and if it is less than the midpoint of the curve, the value is 02.

We have no idea what the midpoint of the curve means. In the python module source code, a modulus operator is used, it simply determines if the first tuple or the y value is odd or even. The multiply tuple contains the x and y coordinates.

If the value in multiply [1] or y is even, then the remainder is 0. The value of the byte variable is 02. If, however, the y value is odd, the modulus operator returns 1 and the string is 3 and not 2. So, it is basically adding a 0 or a 1 to a number, 2. This value 02 or 03 is then added or concatenated at the start to the x value of the point on the Elliptic Curve.

The public key is reduced to nearly half its original size. Given the Elliptic Curve used by Bitcoin and an x coordinate, it is child's play to calculate the y coordinate.

Creating the Field ScriptPublicKey Given a Bitcoin Address

```
ch2108.py
def decode58(string):
    code_string = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
```



```

val = 0
while len(string) > 0:
    val = val * 58
    val = val + code_string.find(string[0])
    string = string[1:]
return val

address = '1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'
aa = decode58(address)
bb = "%x" % aa
print "%s" % bb
bb = bb.decode('hex')
zz = bb[:-4]
print "%s" % zz.encode('hex')
print len(zz)
aa = zz.encode('hex')
script = '76a914' + aa + '88ac'
print script
if script == "76a91457b138abaa1b81a766f18b79b6b2d605ea58dee188ac":
    print "All is good"
else:
    print "vijay Mukhi is an embecile"

```

Output

```

57b138abaa1b81a766f18b79b6b2d605ea58dee11568a07a
57b138abaa1b81a766f18b79b6b2d605ea58dee1
20
76a91457b138abaa1b81a766f18b79b6b2d605ea58dee188ac
All is good.

```

One more important concept here is that given a Bitcoin address, one can automate the script written in the Output ScriptPublicKey Hash field.

Our good old trusty address in the Bitcoin client is

18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7. Please observe the script that gets written in the Outputs section of a Bitcoin transaction.

First caveat, the above code will only work with Bitcoin addresses starting with 1. This base58 string is first converted into a number; to be technically right a base256 number. And the good old decode58 function finishes the job. To convert this number into a string of hex digits, first the %x modifier is used and then the string is decoded. The little that we learnt, the last 8 digits or 4 bytes are for the checksum. The slice operator discards them. The length is 20 bytes now, the RipeMD hash value.

The next task is to simply add the first two opcodes 76 and a9, and then the length of the RipeMD hash value, 0x14 or 20 bytes. The actual hash value is encoded. Finally, the two opcodes, 88 and ac are added at the end.

An encoding is reversible whereas a hash is not.

The EccMultiply function performs the same task as the fast_multiply function. The only change is that it uses Jacobian mathematics which is more difficult for us to understand.

Finding a hmac

```
ch2109.py
import hmac
import hashlib
dig = hmac.new('5678', msg="12345", digestmod=hashlib.sha256).digest()
dig1 = int(dig.encode('hex'),16)
print dig1
```

Output

```
74044394678312947063241410310063013646998902156624128737194809827826706831780
```

hmac stands for hash message based authentication code. In simple words, hmac is a hash value. A message is a Bitcoin transaction and a private key, hashed together. The random number is dependent on both, the private key and the transaction data and it cannot be reversed.

The new function in the module hmac takes a private key and a message hash and it returns a random Sha-256 bit hash value. The code in the pybitcointools/bitcoin library computes a similar hash value. The objective is to stop someone from stealing our nano pennies or Satoshi. But still, our Satoshi's get stolen.

Finally sending a Transaction Using Code Written by Us

```
ch2110.py
import bitcoin
import os
import binascii
import hashlib
#k = 92779573375256004546648151215939474456984759351600370689928618864723097
060474
#priv = "KyUk4a9QDX9CtXsZdnwazDgxbq9V5jmUxQwp3DjvExHmwpHaYmTy"
#priv = "KyH6Mm8sYmnT9cm2jMmxpae7CAvbCmmxQ1bNYsjAeMeeT2XD6KWp"
#priv = "L17qy1UTHX1GTrLHsrRDeFyaaXopVCavGQkQSTK9BrL46Dx4GVP8" # this is very old did not work
Pcurve = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
Gx = 55066263022277343669578718895168534326250603453777594175500187360389116
729240
Gy = 32670510020758816978083085130507043184471273380659243275938904335757337482424
G = (Gx, Gy)
N = 115792089237316195423570985008687907852837564279074904382605163141518161494337
zz = N / 2
print "n/2 = %x" % zz
priv = "L2T9wdbjAvyiXQnm27WPNR323Ce8cDkDkMARQx125CQZ43gHG4yR"
daddr = '1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv'
bitcoins = 55000

def is_bip66(sig):
    sig = bytearray.fromhex(sig)
    print "sig type %s" % type(sig)
    if sig[1] == len(sig)-2:
        sig.extend(b"\1")
```

```

if len(sig) < 9 or len(sig) > 73: return False
if (sig[0] != 0x30): return False
if (sig[1] != len(sig)-3): return False
rlen = sig[3]
if (5+rlen >= len(sig)): return False
slen = sig[5+rlen]
if (rlen + slen + 7 != len(sig)): return False
if (sig[2] != 0x02): return False
if (rlen == 0): return False
if (sig[4] & 0x80): return False
if (rlen > 1 and (sig[4] == 0) and not (sig[5] & 0x80)): return False
if (sig[4+rlen] != 0x02): return False
if (slen == 0): return False
if (sig[rlen+6] & 0x80): return False
if (slen > 1 and (sig[6+rlen] == 0) and not (sig[7+rlen] & 0x80)): return False
return True

def ECadd(x1 , y1 ,x2 , y2):
    m = ( (y2 - y1 ) * inv1( x2 - x1 , Pcurve)) % Pcurve
    x3 = ( m * m - x1 - x2 ) % Pcurve
    y3 = ( m * ( x1 - x3 ) - y1 ) % Pcurve
    return (x3, y3)

def ECdouble(x1 , y1):
    m = ( (3 * x1 * x1 ) * inv1( (2 * y1 ), Pcurve)) % Pcurve
    x3 = (m * m - 2 * x1) % Pcurve
    y3 = (m * ( x1 - x3) - y1) % Pcurve
    return (x3,y3)

def EccMultiply(Gx , Gy , pkey):
    keybin = bin(pkey)[2:]
    x3 , y3 = Gx , Gy
    for i in range (1, len(keybin)):
        #print i , keybin[i]
        x3 , y3 = ECdouble(x3 , y3);
        if keybin[i] == "1":
            x3 , y3 = ECadd(x3 , y3 , Gx , Gy);
    return (x3 , y3)

def reverseahash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def decode58(string):
    code_string = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'
    result = 0
    while len(string) > 0:
        result = result * 58
        result = result + code_string.find(string[0])
        string = string[1:]

```

```
        return result

def inv1(a, n):
    if a == 0:
        return 0
    lm, hm = 1, 0
    low, high = a % n, n
    while low > 1:
        r = high // low
        nm, new = hm - lm * r, high - low * r
        lm, low, hm, high = nm, new, lm, low
    return lm % n

def abc(ind):
    o = ''
    o = o + "01000000"
    o = o + "01"
    inp = thash
    o = o + reversehash(inp)
    outis = "%08x" % outi
    o = o + reversehash(outis)
    if ind == 0:
        scrlen = len(addr1)/2
        scrlen = "%02x" % scrlen
        o = o + scrlen
        o = o + addr1
    else:
        scrlen1 = len(dummy3)/2
        scrlen2 = "%02x" % scrlen1
        o = o + scrlen2
        o = o + dummy3
        o = o + "ffffffff"
        o = o + "01"
        zz = "%016x" % bitcoins
        zz = reversehash(zz)
        o = o + zz
        scrlen1 = len(addr2)/2
        scrlen1 = "%02x" % scrlen1
        o = o + scrlen1
        o = o + addr2
        o = o + "00000000"
    return o

pub = bitcoin.privkey_to_pubkey(priv)
address = bitcoin.pubkey_to_address(pub)
print "source bitcoin address=%s" % address
addr1 = bitcoin.mk_pubkey_script(address)
print "addr1=%s" % addr1
```

```

unspend = bitcoin.history(address)
unspendl = []
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)
if len(unspendl) == 0:
    print "No balances to spend"
    exit(0)
spendd = unspendl[0]
print "Total Number are %d" % len(unspend)
print "Number of unsent outputs are %d" % len(unspendl)
print "spendd=%s" % spendd
newobject = spendd['output']
print "newobject=%s" % newobject
ind = newobject.index(':')
thash = newobject[0:ind]
outi = int(newobject[ind + 1:])
addr2 = bitcoin.mk_pubkey_script(daddr)
print "addr2=%s" % addr2
signing_tx = abc(0)
print "signing_tx=%s" % signing_tx
dummy = hashlib.sha256(hashlib.sha256(signing_tx.decode('hex') + "\x01\x00\x00\x00").digest()).digest()
z = int(dummy.encode('hex') , 16)
k = bitcoin.deterministic_generate_k(dummy, priv)
print "Random Number k=%s" % k
r, y = EccMultiply(Gx, Gy, k)
dummy10 = decode58(priv)
dummy10 = ("%x" % dummy10).decode('hex')
print "dummy10=%s" % dummy10.encode('hex')
dummy1 = dummy10[1:-4][:32]
print "dummy1= %s" % dummy1.encode('hex')
decodeprivkey = int(dummy1.encode('hex'),16)
s = inv1(k, N) * (z + r * decodeprivkey) % N
print "s=%x" % s
s = N - s if s > N // 2 else s # BIP62 low s
print "s=%x" % s
r1 = ("%064x" % r).decode('hex')
s1 = ("%064x" % s).decode('hex')
print "before r1=%s" % r1.encode('hex')
print "before s1=%s" % s1.encode('hex')
print "First byte of r1 %x" % bytearray(r1)[0]
print "First byte of s1 %x" % bytearray(s1)[0]
if bytearray(r1)[0] & 0x80:
    print "b1 negative"
    r1 = b'\x00' + r1
if bytearray(s1)[0] & 0x80:
    print "b2 negative"

```

```
s1 = b'\x00' + s1
print "After r1=%s" % r1.encode('hex')
print "After s1=%s" % s1.encode('hex')
lenr1 = ("%02x" % len(r1)).decode('hex')
lens1 = ("%02x" % len(s1)).decode('hex')
print "lenr1=%s" % lenr1.encode('hex') , "lens1=%x" % int(lens1.encode('hex'),16)
left = b'\x02' + lenr1 + r1
print "left =%s" % left.encode('hex')
right = b'\x02' + lens1 + s1
print "right=%s" % right.encode('hex')
totallength = ("%02x" % len(left+right)).decode('hex')
print "Total length %d:%x" % (len(left+right),len(left+right))
sighex = (b'\x30' + totallength + left + right).encode('hex')
print "sighex=%s" % sighex
sig = sighex + '01'
print "sig= %s" % sig
assert is_bip66(sig)
lensig = len(sig)/2
lenpub = len(pub)/2
print "lensig=%0x" % lensig
print "lenpub=%0x" % lenpub
dummy3 = ("%02x" % lensig + sig + "%02x" % lenpub + pub
tx2 = abc(1)
os.system("bitcoin-cli decoderawtransaction " + tx2)
os.system("bitcoin-cli sendrawtransaction " + tx2)
```

This is the output for a transaction run in January 2017.

Output

```
n/2 = 7ffffffffffffffffffffffffffffffff5d576e7357a4501ddfe92f46681b20a0
source bitcoin address=1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n
addr1=76a914d7ce0c2c237ec0ec04931335377a87d16b9865ad88ac
Total Number are 37
Number of unspent outputs are 4
spendd={'output':
u'6a5b95274f01722c52e4a1f76a4c161b9a3d42c3c22afe6b01a60760d52566b0:0',
'block_height': 447289, 'value': 69700, 'address': u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}
newobject=6a5b95274f01722c52e4a1f76a4c161b9a3d42c3c22afe6b01a60760d52566
b0:0
addr2=76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac
signing_tx=0100000001b06625d56007a6016bfe2ac2c3423d9a1b164c6af7a1e4522c72
014f27955b6a000000001976a914d7ce0c2c237ec0ec04931335377a87d16b9865ad88ac
ffffffff017c150000000000001976a914c847b6d84ecf8048474b19c4a2330409ff32a1ad8
8ac00000000
Random Number
k=765590715687455061954701891865802251289734208759242913484658168099966
5205983
```

```

dummy10=809c1a676959bcbd5568a1ea251a65a01e03fd3866e61591558abb0157aa14
8324017c9136ec
dummy1=
9c1a676959bcbd5568a1ea251a65a01e03fd3866e61591558abb0157aa148324
s=89492520204e685bfc468a0b5027451576a5bdee7c6faed335775fda3f4fb2a7
s=76b6dadfd9b197a403b975f4afd8bae944091ef832d8f1688a5afeb290e68e9a
before
r1=55a3097dc0a3dbd46749c45d7a018589062ed06790c0ae4f6199743209c49561
before s1=76b6dadfd9b197a403b975f4afd8bae944091ef832d8f1688a5afeb290e68e9a
First byte of r1 55
First byte of s1 76
After r1=55a3097dc0a3dbd46749c45d7a018589062ed06790c0ae4f6199743209c49561
After s1=76b6dadfd9b197a403b975f4afd8bae944091ef832d8f1688a5afeb290e68e9a
lenr1=20 lens1=20
left
=022055a3097dc0a3dbd46749c45d7a018589062ed06790c0ae4f6199743209c49561
right=022076b6dadfd9b197a403b975f4afd8bae944091ef832d8f1688a5afeb290e68e9a
Total length 68:44
sighex=3044022055a3097dc0a3dbd46749c45d7a018589062ed06790c0ae4f619974320
9c49561022076b6dadfd9b197a403b975f4afd8bae944091ef832d8f1688a5afeb290e68e
9a
sig=
3044022055a3097dc0a3dbd46749c45d7a018589062ed06790c0ae4f6199743209c4956
1022076b6dadfd9b197a403b975f4afd8bae944091ef832d8f1688a5afeb290e68e9a01
sig type <type 'bytearray'>
lensig=47
lenpub=21
{
  "txid": "0e4a403f41837a561b56bb7aa1d6518fecb9f4f9d6718b56f728488f34620063",
  "hash":
    "0e4a403f41837a561b56bb7aa1d6518fecb9f4f9d6718b56f728488f34620063",
  "size": 191,
  "vsize": 191,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid":
        "6a5b95274f01722c52e4a1f76a4c161b9a3d42c3c22afe6b01a60760d52566b0",
      "vout": 0,
      "scriptSig": {
        "asm":
          "3044022055a3097dc0a3dbd46749c45d7a018589062ed06790c0ae4f6199743209c4956
1022076b6dadfd9b197a403b975f4afd8bae944091ef832d8f1688a5afeb290e68e9a[ALL
]038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3",
        "hex":

```

```
        "473044022055a3097dc0a3dbd46749c45d7a018589062ed06790c0ae4f6199743209c49
        561022076b6dadfd9b197a403b975f4afd8bae944091ef832d8f1688a5afeb290e68e9a01
        21038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3"
    },
    "sequence": 4294967295
  }
],
"vout": [
  {
    "value": 0.00005500,
    "n": 0,
    "scriptPubKey": {
      "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
      OP_EQUALVERIFY OP_CHECKSIG",
      "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
      "reqSigs": 1,
      "type": "pubkeyhash",
      "addresses": [
        "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
      ]
    }
  }
]
}
```

Output

0e4a403f41837a561b56bb7aa1d6518fecb9f4f9d6718b56f728488f34620063

For comparison, the same program run in the year July 2016. having over 28000 confirmations.

$n/2 = 7$ fffffffffffffffffffffffffffffffff5d576e7357a4501ddfe92f46681b20a0

source bitcoin address=1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n

Total Number are 9

Number of unsent outputs are 5

spendd={'output':

u'8c408645b608c2b4c7d3977c2e2ed7b1051aa646f62b0bc943e529fbaf2f8304:1',

'block_height': 419250, 'value': 64300, 'address':

u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}

Random Number

k=103197897402501968834016053256061124556353834128760791741409176712008

630576899

s=9210371331650156247692863812223680274351760224316895188839761908033301

2303293

s=2368837592081463294664234688645110510931996203590595249420754406118514

9191044

r1=ab48fe68c49437562dda71d9ace04410e3e7087c221c6814876a3596564e11bb

s1=345f2548a71ed928c17c38584cbde09f6150d54a17fe0bc2efe76ee57f814384


```

b1 negative
lenr1=21 lens1=20
left
=022100ab48fe68c49437562dda71d9ace04410e3e7087c221c6814876a3596564e11bb
right=0220345f2548a71ed928c17c38584cbde09f6150d54a17fe0bc2efe76ee57f814384
Total length 69:45
sighex=3045022100ab48fe68c49437562dda71d9ace04410e3e7087c221c6814876a359
6564e11bb0220345f2548a71ed928c17c38584cbde09f6150d54a17fe0bc2efe76ee57f8
14384
sig=3045022100ab48fe68c49437562dda71d9ace04410e3e7087c221c6814876a359656
4e11bb0220345f2548a71ed928c17c38584cbde09f6150d54a17fe0bc2efe76ee57f8143
8401
sig type <type 'bytearray'>
lensig=48
lenpub=21
{
  "txid": "e57b2fab5debba6d4a28616d7d07ca030fe7b0584a5db14f1c6514d5c6d1fd14",
  "size": 192,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid":
      "8c408645b608c2b4c7d3977c2e2ed7b1051aa646f62b0bc943e529fbaf2f8304",
      "vout": 1,
      "scriptSig": {
        "asm":
        "3045022100ab48fe68c49437562dda71d9ace04410e3e7087c221c6814876a3596564e1
        1bb0220345f2548a71ed928c17c38584cbde09f6150d54a17fe0bc2efe76ee57f814384[A
        LL] 038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3",
        "hex":
        "483045022100ab48fe68c49437562dda71d9ace04410e3e7087c221c6814876a3596564
        e11bb0220345f2548a71ed928c17c38584cbde09f6150d54a17fe0bc2efe76ee57f81438
        40121038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00055000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
        OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",

```

```
        "reqSigs": 1,  
        "type": "pubkeyhash",  
        "addresses": [  
            "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"  
        ]  
    }  
}  
]  
}  
e57b2fab5debba6d4a28616d7d07ca030fe7b0584a5db14f1c6514d5c6d1fd14
```

We have finally reached the end of a long arduous journey. In the above program, we have sent out a transaction using code mainly written by us. Though, there is some code taken from the Bitcoin and pybitcointools/bitcoin modules, but this code has been explained in the past.

We start from the first lines of our code. It starts with our usual suspects, the imports. Only four of them. Then we have the four standard Elliptic Curve parameters.

We must have run this program over 100 times, checking and rechecking, so that it works as advertised. We paid good money for the Bitcoins, though all got used up while trying our code or they got stolen. Yes stolen.

We did a very dumb thing, we merrily used the same number for random number holding variable k (it has been commented out in our code now) with our old private key starting with KyU. So, all Bitcoins associated with this private key got spent without our knowledge.

With thoughts that someone hacked into our iMac, we created another Bitcoin address with a private key starting with KyH. All was well and suddenly these Bitcoins also got stolen. After a good night's sleep, it dawned upon us that there could be a possibility that someone was reading our input signatures and checking if the same random number was used.

To reconfirm, we sent a few Bitcoins to a Bitcoin address:

18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7 and the Bitcoins were claimed in seconds.

The third private key is a very old one but for some reason, the wallet provider Zebpay refused to send any Bitcoins to it.

The latest trend in cybercrime is to steal Bitcoins as it is next to impossible to find out who stole them. The people who were associated with Bitcoin mining in the early days of Bitcoins are been personally targeted. People analyze each Bitcoin transaction. Most people use a wallet to send a Bitcoin transaction.

The company Zebpay have our private keys, and the fear is they can go rogue, as well. So, please chose your wallet provider with utmost care. It is your money at stake. Play safe by using the largest wallet companies only. The wallet company can also get hacked and your private keys stolen. If your private key gets stolen, welcome to the club.

One of the many reasons for explaining the hmac program earlier.

The glitch in the program is that the private keys are revealed, so none of these Bitcoin addresses will have any money left in them when we publish this book. The variable priv stores the same private keys used in the past. The code is executed with the same private key at two different times to show the differences in output.

The destination Bitcoin address is stored in variable, daddr. Please, please replace this with your own Bitcoin address or else our address will receive the bitcoins. This is one Bitcoin address we own but we do not have the private key of this Bitcoin address but we can yet say that it is our Bitcoin address. Also, it is highly recommended that you change the private key, priv to a Bitcoin address that you own and make sure it has some Bitcoins.

The value for the Bitcoins to be transferred is 5500 Satoshi. The variable used to store these Bitcoins is aptly named `bitcoins`. The Bitcoin address starts with 1Lg which we calculate from the private key. The value initially transferred was Rs. 30 per transaction in Indian currency and not Satoshis. Subsequently, the value was raised to Rs. 50 as the price of Bitcoins doubled. Each time the code is executed, one unspent output of the transaction gets used up. Now the amount to spend per transaction is multiplied by 10, i.e. Rs 500 per transaction. We prefer the testnet.

The program code is simplified by using the entire Bitcoin amount in one go. Since we own both the Bitcoin addresses, there is no question of sending part money to another Bitcoin address and part money back to our Bitcoin address.

When miners get extremely busy, they start asking for more money to add our transaction in a block. That's why an error is displayed, which reads insufficient funds. Well there is no need to panic, simply reduce the number of bitcoins by one less 0, i.e. 550. This has always worked for us.

The Bitcoin community decided to make signature verification more stringent, so a new function called `is_bip66` is added. This Bitcoin Improvement Protocol number 66 has these new rules and guidelines. More on this function later, it only does an error check and thereafter, it returns True or False.

There are a set of three functions that does the Elliptic Curve multiplication, one is `EccMultiply`. This function in turn calls functions `ECadd` and `ECdouble`. This is explained in one of the earlier chapters but we did not remember from where this code came from. Then, there is function `reversehash` that reverses an encoded string or hash the Bitcoin way. A base58 encoded string is converted to a base256 number using the `decode58` function. Thereafter comes the modular inverse function `inv1`. It was and still is difficult to explain this function as there are too many calculations and we never liked math as such.

The second new function introduced in the program is function `abc`. At one level, this function only returns an encoded string. It represents the raw bytes of part of a transaction.

Back to code.

To make life easier, the variable `pub` carries the calculated public key using the standard Bitcoin module function `privkey_to_pubkey`. This is followed by the function `pubkey_to_address`. In the end, we get the Bitcoin address associated with this public key.

Earlier, a variable was assigned a hardcoded Bitcoin address. In this program, the private key starting with L2T gives us the public key from where a Bitcoin address is dynamically calculated. Please make sure there is some money in the Bitcoin address starting with 1Lg first. This Bitcoin address, stored in variable `address` is however not used other than the `history` function.

The output section of the transaction hash uses some opcodes + the RipeMD 160-byte hash + more opcodes. The function `mk_pubkey_script` gives us a usable script value in variable `addr1`. This variable will be used later when the raw bytes of the transaction hash are created. Once again, output displayed confirms that the private key starting with L2T9 gives a Bitcoin address starting with 1Lg5.

The variable `addr1` starts with the normal opcodes, 76 and a9. This is followed by the length of the RipeMD hash value, 20 and then the actual hash bytes. At the end of the script, there are the familiar opcodes, 88 and ac.

The next couple of lines of code use the good old `history` function. The task is to find an unspent output in the Bitcoin address calculated from the private key. The size of variable `unspend` could be 34 or anything. We must have unspent outputs, that's all.

The Bitcoin address is associated with a set of transactions and the spent outputs must be removed from it. The `for` loop stores a list, `unspendl` for all those records that do not have a spend key. The output with this key is not useful for us. The size of the `unspendl` list determines the number of outputs to spend. The size of this list is only 4, your mileage will vary.

There are not too many outputs to use. We started with 36 unspent transactions outputs at one time and now we are left with only 4 unspent outputs. The next time we run this program we will have only 3 unspent outputs. Once again, in the for loop each of these transactions are scanned to check for a key called spend.

The history function uses this key and determines who has spent the output. The transactions that do not have the spend key is added to the unspendl list and others are ignored. There is an error check here. If there are no outputs to spend, we have no Bitcoins associated with this Bitcoin address, we have no choice but have the program quit out.

The variable spendd takes up the first row of the unspendl list. It's been displayed to cross check with the blockchain explorer by you. The key output and the address is what the doctor ordered. But this output is now spent.

The variable newobject is being initialized by a known key called output. This is very familiar, a transaction hash value, a colon and an index output number. We do the normal shenanigans and extract the output index in variable outi and the transaction hash value in variable thash. There is a transaction hash value and output index of the output bearing Bitcoins. The variable addr2 is the script public key value of the destination Bitcoin address stored in variable daddr. This comprises the ripemd hash and the standard opcodes. This Bitcoin address starts with 1KFz.

The function abc is called with 0 as a parameter. This is where the address addr1 comes in. There is an empty string o. To the string, first the version number 1 is added and it takes up 4 bytes, so the value is 01000000 and not 01. The various fields of a transaction are concatenated to the string o, which is eventually returned.

Then comes the number of inputs, 1 or 01. Most numbers use a variable number of bytes to hold their values. Now starts the single input. The hash value of the transaction containing the output is reversed. This transaction hash value is stored in the variable thash. The output index is converted into a number, 8 hex digits large or 4 bytes. This output index is stored in the variable outi.

The length is a fixed 4 bytes but 8 hex digits. Because of the endianness, the bytes are reversed again. The function is called reversehash, but ideally it should be called reverse the endianness.

Now for explaining the need of a parameter to the abc function.

When a transaction is signed, a script public key is placed in the inputs. After the signature is calculated, the signature + public key is placed in the same field. This is the Bitcoin law.

So, in the same input field, the value of addr1 or the actual signature plus public key (computed later) is placed. However, the length is placed first and then the script. As the parameter ind is 0, the length of the script stored in variable addr1 is determined. It is divided by 2 as it is an encoded string. Then there is an addition of the length and the opcodes + ripemd hash + opcodes stored in variable addr1. Finally, what is obtained is the last field of the inputs, the unused sequence number, and all F's.

Now start the outputs. First is the total number of outputs, 1. The bitcoins variable holding the Bitcoin value is 16 hex digits or 8 bytes large. The string is reversed, as always, for all the number strings.

Next comes the destination script, which is present in variable addr2 as opcode + hash + opcodes. It is first prefaced with a length which is a known constant. The length is determined and then divided by 2. Finally, the last field lock time, rarely used, is all 0's. The return value is stored in string o and it must be signed later. But first, it must be decoded and then the double Sha-256 hashed.

Wait a second, a number 1 is appended, but in little endian. This number is also known as a constant SIGHASH_ALL in the Bitcoin literature.

The variable dummy is a Sha-256 hash value; this decoded string is converted into a number using the int function. The variable z stores this dummy hash as a number.

A random number k is calculated using the function deterministic_generate_k. This function is given two unique seeds.

These values depend upon the transaction data, which is the transaction hash stored in variable `dummy` and the private key which will also be unique per user.

The transaction hash is dependent upon the transaction data and it will change for every transaction. The private key should also change per transaction. As per convention, never reuse a Bitcoin addresses and hence the associated private key.

The generator point `G` is multiplied with the private key `k`. Either of the two functions, `fast_multiply` or `EccMultiply` can be used. Both use arguably a different method to multiply a point on the Elliptic Curve with a number. A tuple is returned which represents the `x` and `y` points but by convention, we call them `r` and `y`. If the variable `k` was a private key and not a random number, then we could have calculated the public key.

Earlier while understanding the module `pybitcointools` or `bitcoin` code, we hard coded the values of `r` and `s` and hence we lost bitcoins. The variables `Gx` and `Gy` are a constant thanks to the `secp256k1` standard. The variable `k` we chose to make it a constant. Therefore, the variables `r` and `s` were also a constant.

Feel free to print out the values of variables `r` and `s` and with a warning here, send some transactions with these variables `r` and `s` having constant values. Your Bitcoin address will magically get robbed of all its Bitcoins.

There are a series of variable names that make no sense. This is only because these variables values are temporary. The variables names do not make sense, their values, however do.

There is a private key in variable `priv` that is in a `wif_compressed` format. This format may be smaller, full of an error checks but the requirement is the original undiluted number. This variable is created by the wallet. The function `decode58` gives a number from a string which is then converted into a hex string using the `%x` modifier and stored in variable `dummy10`. The string is then decoded and stored back into variable `dummy10`.

Assuming this is a `wif_compressed` private key, we must consider the extra `01` at the very end. Also, the first byte `80` and the last four checksum bytes are removed. The length is reduced to 32 bytes using the slice operator `[:32]`. Too many slice operators on one line. This variable that contains a smaller string is called `dummy1`. It's at moments like this when we use meaningless variable names starting with `dummy`. It will be worth watching the output of variables `dummy10` and `dummy1`. What you will visually see are the bytes that have been removed. Finally, the encoded private key is converted to an int and the variable name used is `decodeprivkey`.

For signing, the same modular inverse function `inv1` is used. It is given the random number variable, `k` and the `secp256k1` constant, `N`. This mod inverse is multiplied with three values. First is the variable `z`, which is the transaction hash value as a number. The second one is `r`, which is the `x` value of a point in an Elliptic Curve and third is the newly calculated modified private key (as in number) stored in variable `decodeprivkey`.

The value must never exceed the value of `N`.

According to BIP 62, not to be confused by BIP 66, the signature `s` must have a value between `0x1` and

`0x0x7FFFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0`.

This value is obtained by dividing the value in variable `N` by 2. For good luck, the value in the variable `zz` whose value is `N/2` is displayed at the start of the code. Both are the same.

According to BIP 62, if the value of `s` is higher than `N/2`, then it must be reduced to `N - s`. If not then the value of `s` is unchanged. This is important otherwise the method `sendrawtransaction` will not send this transaction or the miner will reject it. Sometimes the transactions will go through but then the error check steps in. Again, if the value in `s` is in range, the value remains the same.

The next step is to concatenate the signature and the public key together. Our signature is DER encoded thus all the rules of DER must be followed. The encoding scheme called DER invented by the Europeans is very old. The full form

is Distinguished Encoding Rules and its other brother is called BER. The 1 byte sighash flag added at the end is however not part of DER encoding. There is a separate BIP called 66 that places lots of stringent checks. The BIP 66 basically is source code of a C function that implements different error checks. The pybitcointools/bitcoin library simply converts these C error checks into Python.

But let's not digress. To create a signature, the value in `r` and `s` is converted into hex digits, `r1` and `s1`. As they are strings, the first byte is checked. Bitwise anding with a 1 gives 1, otherwise 0. If the top most bit of the first byte is a 1, an extra 0 is added at the beginning of the `r` and `s` variables. When we bitwise and something with 0x80, we are setting all the other bits to 0 but leaving the 7th bit as is, where is. This has something to do with the interpretation of `r` and `s` as negative numbers. The byte array function is used to gain access to the individual bytes.

No such luck in our case. The `r1` and `s1` variables look and feel the same. No change.

Two more variables, `lenr1` and `lens1` are created which are given the length of the two strings, `r1` and `s1`. The strings are converted into hex digits using the `%x` modifier. They are then decoded. Their values can be displayed only after encoding. The length of the `lenr1` and `lens1` variables are 20 hex or 32 bytes as a string, however they may increase to 0x21.

The variable `left` has three entities. First a value of 02 is added, followed by the length of the string stored in variable `lenr1` which is 20 and then the actual point `r1`. All of them are in string format. The variable `right` is computed the same way as the variable `left`, the `r1` variable is replaced by `s1`. These variables are called `left` and `right` only because the `r` value comes first followed by the `s` value. All this is created dynamically.

The variable `totallength` stores the length of the variables, `left` and `right`. These variables are determined from the values of variables, `r1` and `s1`. The length is the sum of the individual lengths, in our case 0x44 to 0x46 bytes.

The final signature is made up in the following way. It starts with the DER byte 0x30, then the total number of bytes following, 0x44 or the `totallength` variable, then the values of `r` and `s`, which are stored in the variables `left` and `right`. We should have called `right` and `left` variables as signature and public key.

This procedure is adopted to get the signature and public key in the right format. To simplify things, we could simply hard code the signature and public key and added the lengths and other DER stuff.

Now this is not the end of the story as the next task is to add the sighash value, 01 at the end of the above string. This variable is called `sig`. Moreover, the signature stored in variable `sig` must be BIP 66 compliant.

The length of the signature and the length of the public key are placed in two variables, `lensig` and `lenpub`. The value in `lensig` will be 47 or 48 hex bytes and the value in `lenpub` will be 0x21. We will disclose in the last chapter what the actual signature lengths can be. The final variable `dummy3` is created which starts with the length of the signature, as in the value in variable `lensig`. Then the entire signature is written and then comes the length of the public key, i.e. the value in variable `lenpub`. And finally, the actual public key.

When the function `abc` is executed with a value other than 0, the process differs only a little. First the length of this script stored in variable `dummy3` is computed in variable `scrlen1`. It is the length divided by 2. The variable `scrlen1` is used to store the length of the entire signature+public key. The variable `scrlen2` has the same value as variable `scrlen1` but in a string format. The script length is followed by the actual script, which is stored in `dummy3`.

Now let's check if the rules laid out in BIP 66 are followed. There are more error checks other than this BIP, as well. If our transaction is sent to a miner who follows this BIP, he/she will not reject it. We have an entire chapter in the end which looks at transaction error checks.

The function `is_bip66` is passed an encoded string. Had we used the function `decode` to form a decoded string, the `ord` function would be used next to extract one byte from the string at a time. A better way out here is to convert the string into a bytearray using the `fromhex` function of the `bytearray` module.

Now let's ensure that the last byte of the signature has a value of sighash or 01. The value in the last byte must be checked as signatures + public keys, they can also end with 01. The second byte sig[1] is the length of the signature. If the signature length which is len(sig) - 2 is equal to this value sig[1], a 01 is added at the end of the signature. This is not an error check.

All the error checks will come next and a value of false is returned if the error check fails. The assert screams blue murder if the return value of the function is false.

Error check #1. The signature size must be greater than 9 and less than 73.

Error check #2. The first byte of the signature must be 0x30. DER calls this a compound entity. A DER rule to be obeyed.

Error check #3. The length of the signature stored in byte sig[1] must be equal to the length of the signature - 3. There is a very valid reason for doing this. 3 is deducted to account for the 0x30, the first DER byte, the signature length and the last byte 01.

A helper variable rlen is created to store byte 3, i.e. sig[3], following the signature byte length.

Error check #4. The s element must be within the signature bounds. We could extract the s length and then check that the length of the two elements are within the signature. Instead we add 5 to the length of the r element i.e. rlen and check if this is greater than the signature length. Thereafter the length byte of the s element is determined. This value is stored in the variable slen. This value is stored at 5 bytes + the length of the r element. It must be noted that the length of r element itself starts 3 from the start.

Error check #5. Now the lengths of the r and s elements stored in variables rlen and slen plus 7 must be equal to the length of the string.

Error check #6. The first byte of the r element or sig[2] must be 02 as per the DER specifications for an integer.

Error check #7. The length of r or variable rlen must not be 0, so the length is checked to be 0.

Error check #8. The value of r must be a positive number so the value in the first real byte of r which is sig[4] is checked. Its uppermost bit must not be 1, also when bitwise anding with 0x80.

Error check #8. There are multiple checks in one if. The value in r cannot be a null byte. This check is performed by making sure that the length of r is greater than 1, the fourth byte of the signature is 0 and the next byte following must not have its high bit set. This has all to do with negative numbers. The process for r is repeated for s.

Error check #9. The first byte of s, sig[4 + rlen] is checked to have a value of 0x02.

Error check #10. The length of s must not be zero, like r, so a check is performed on the length of s.

Error check #11. The second last check for s is that like r, s cannot be a negative number.

Error check #12. Finally, like r, s cannot have null bytes at the start.

The fascinating thing about the BIP 66 is that it has comments for every check, unlike most of the code we have seen in python modules.

When you have nothing else to do please try out the following transaction hashes, they will reveal how many times we executed the above program.

```
a23dbea98a80b921d9b4ca5cc3c82a21b29a13b53e92d1f6c6d64ab3a48eb5d5
ec253f734647dd52a60b62ffb1d0271ec592696adb241be31a956b07081c2178
```

```
One type of error
error code: -26
```

error message:
258: txn-mempool-conflict

The above error is a common one and it pops up many a times while executing the program. The mempool is an area of memory where the Bitcoin peer stores all incoming unconfirmed transactions. When a transaction is confirmed in a block, these transactions get removed from the mempool. The mempool is a dynamic staging area for transactions. Every mempool in every node will have a different set of transactions.

Though we remain clueless on the reason behind this error, it was resolved by stopping the Bitcoin daemon, and then restarting the Bitcoin Core GUI. A wait of 5 minutes, shut down the GUI, started the daemon and everything works magically.

error code: -26
error message:
66: insufficient priority

The error message of insufficient priority occurs when the miner is not paid enough money for his/her efforts. Reduce the bitcoins variable value to 550, a very small amount. The transaction goes through but now a new error shows up in the blockchain explorer.

Transaction rejected by our node. Reason: All Outputs Are Very Small you must include a min (10000) Satoshi fee

Some more transaction hash values for you to check:

848fc340ff51a4f9f1119e330039f8d19bf993a14e50f895fefbb2c082af94d2
ec78016740f34beea68dea7b8448942ebb8afeb7a55354eb5c89508fd3533a9c

This will convince you that we have spent lots of Bitcoins in making sure that our code works.

We are convinced that code that we write will never ever work. So, on the 12th of October 2017 for the nth time, the transaction id we get is:

de849b5426aede2db21aac7a5a1d5b625ff7f7ab4825136f2415484794bd417a

CHAPTER 22

Sending One Transaction Without Using Library Functions

This chapter has only one program which brings together all the fragmented Python code written in various chapters. Though we hate books with large programs, we had to put it all together some day, so here it is.

The program has very few ready-to-use library functions; most of the functions are user-defined. For example, in place of `sendrawtransaction`, we use our own code which was explained in the second networking chapter. Similarly, we have user-defined functions to convert a private key into a Bitcoin address. There are no imports for `bitcoin` or `pybitcointools` module, even no `hashlib`. Only standardized libraries from Python are used.

The code with the `Sha256` hash is improvised, as earlier it worked with a very small message size. The code has one extra loop added. The computation of the `RipeMd-160` hash value is also updated as the hash type requires the size of the data at the very end.

A few more changes are incorporated like function names are changed from `make_request` to `make_request1` and `history` to `history1`. The connection is to our local bitcoin server `bitcoind` so we take the liberty to play around with the networking principles and some other values in different fields. Though we agree that we have been too lazy to change all the variable names and the format of the code.

Sending a Transaction to a Miner With Our Code in Place of Library Functions

```
ch2201.py
import json
import urllib2
import random
import re
import struct
import socket
import time
import os
#You must change this value to a wif_compressed private key created by the bitcoin-cli
priv = "L2T9wdbjAvyiXQnm27WPNR323Ce8cDkDkMARQx125CQZ43gHG4yR"
#If you do not change this to your Bitcoin address, I will receive Bitcoins
#send the bitcoins to me. This is allowed
04ffb821f8cef805d104615333c6e6758601fdeb1a6f48fd6621595915cbe38c
daddr = '1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv'
#This is optional as it is a very low number in Satoshi
bitcoins = 5500
```

```
Pcurve = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
Gx = 55066263022277343669578718895168534326250603453777594175500187360389116
729240
Gy = 32670510020758816978083085130507043184471273380659243275938904335757337
482424
G = (Gx, Gy)
N =
115792089237316195423570985008687907852837564279074904382605163141518161494337

rho = [7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8]
pi = [5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12]
rl = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], [7, 4, 13, 1, 10, 6, 15, 3,
12, 0, 9, 5, 2, 14, 11, 8], [3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12], [1, 9,
11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2], [4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8,
11, 6, 15, 13]]
rr = [[5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12], [6, 11, 3, 7, 0, 13, 5, 10,
14, 15, 8, 12, 4, 9, 1, 2], [15, 5, 1, 3, 7, 14, 6, 9, 11, 8, 12, 2, 10, 0, 4, 13], [8, 6,
4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14], [12, 15, 10, 4, 1, 5, 8, 7, 6, 2, 13, 14,
0, 3, 9, 11]]
f1 = lambda x, y, z: x ^ y ^ z
f2 = lambda x, y, z: (x & y) | (~x & z)
f3 = lambda x, y, z: (x | ~y) ^ z
f4 = lambda x, y, z: (x & z) | (y & ~z)
f5 = lambda x, y, z: x ^ (y | ~z)
fl = [f1, f2, f3, f4, f5]
fr = [f5, f4, f3, f2, f1]
sl = [[11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8], [7, 6, 8, 13, 11, 9, 7, 15,
7, 12, 15, 9, 11, 7, 13, 12], [11, 13, 6, 7, 14, 9, 13, 15, 14, 8, 13, 6, 5, 12, 7, 5],
[11, 12, 14, 15, 14, 15, 9, 8, 9, 14, 5, 6, 8, 6, 5, 12], [9, 15, 5, 11, 6, 8, 13, 12, 5,
12, 13, 14, 11, 8, 5, 6]]
sr = [[8, 9, 9, 11, 13, 15, 15, 5, 7, 7, 8, 11, 14, 14, 12, 6], [9, 13, 15, 7, 12, 8, 9,
11, 7, 7, 12, 7, 6, 15, 13, 11], [9, 7, 15, 11, 8, 6, 6, 14, 12, 13, 5, 14, 13, 13, 7, 5],
[15, 5, 8, 11, 14, 14, 6, 14, 6, 9, 12, 9, 12, 5, 15, 8], [8, 5, 12, 9, 12, 5, 14, 6, 8,
13, 6, 5, 15, 13, 11, 11]]
KL = [0, 0x5a827999, 0x6ed9eba1, 0x8f1bbcdc, 0xa953fd4e]
KR = [0x50a28be6, 0x5c4dd124, 0x6d703ef3, 0x7a6d76e9, 0x0]
def rol(s, n):
    return ((n << s) | (n >> (32-s))) & 0xffffffffL
def box(h, f, k, w, r, s):
    (a, b, c, d, e) = h
    for word in range(16):
        T = (a + f(b, c, d) + w[r[word]] + k) & 0xffffffffL
        T = (rol(s[word], T) + e) & 0xffffffffL
        (b, c, d, e, a) = (T, b, rol(10, c), d, e)
    return (a, b, c, d, e)
def ripemd160(s):
    hl = hr = h = (0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476, 0xc3d2e1f0)
```

```

w = struct.unpack("<16L", s)
for round in range(5):
    hl = box(hl, fl[round], KL[round], w, rl[round], sl[round])
    hr = box(hr, fr[round], KR[round], w, rr[round], sr[round])
    h =
    ((h[1]+hl[2]+hr[3])&0xFFFFfffL,(h[2]+hl[3]+hr[4])&0xFFFFfffL,(h[3]+hl[4]+hr[0])&0
    xFFFFfffL,(h[4]+hl[0]+hr[1])&0xFFFFfffL,(h[0]+hl[1]+hr[2])&0xFFFFfffL)
    return struct.pack("<5L", *h).encode('hex')

k = (0x428a2f98L, 0x71374491L, 0xb5c0fbcfL, 0xe9b5dba5L, 0x3956c25bL,
0x59f111f1L, 0x923f82a4L, 0xab1c5ed5L, 0xd807aa98L, 0x12835b01L, 0x243185beL,
0x550c7dc3L, 0x72be5d74L, 0x80deb1feL, 0x9bdc06a7L, 0xc19bf174L, 0xe49b69c1L,
0xefbe4786L, 0x0fc19dc6L, 0x240ca1ccL, 0x2de92c6fL, 0x4a7484aaL, 0x5cb0a9dcL,
0x76f988daL, 0x983e5152L, 0xa831c66dL, 0xb00327c8L, 0xbf597fc7L, 0xc6e00bf3L,
0xd5a79147L, 0x06ca6351L, 0x14292967L, 0x27b70a85L, 0x2e1b2138L, 0x4d2c6dfcL,
0x53380d13L, 0x650a7354L, 0x766a0abbL, 0x81c2c92eL, 0x92722c85L, 0xa2bfe8a1L,
0xa81a664bL, 0xc24b8b70L, 0xc76c51a3L, 0xd192e819L, 0xd6990624L,
0xf40e3585L, 0x106aa070L, 0x19a4c116L, 0x1e376c08L, 0x2748774cL,
0x34b0bcb5L, 0x391c0cb3L, 0x4ed8aa4aL, 0x5b9cca4fL, 0x682e6ff3L, 0x748f82eeL,
0x78a5636fL, 0x84c87814L, 0x8cc70208L, 0x90befffaL, 0xa4506cebL, 0xbef9a3f7L,
0xc67178f2L)
_h = (0x6a09e667L, 0xbb67ae85L, 0x3c6ef372L, 0xa54ff53aL, 0x510e527fL,
0x9b05688cL, 0x1f83d9abL, 0x5be0cd19L)

def _rotr(x, y):
    return ((x >> y) | (x << (32-y))) & 0xFFFFFFFFL

def sha256(data):
    bytes = ""
    h0 = 0x6a09e667
    h1 = 0xbb67ae85
    h2 = 0x3c6ef372
    h3 = 0xa54ff53a
    h4 = 0x510e527f
    h5 = 0x9b05688c
    h6 = 0x1f83d9ab
    h7 = 0x5be0cd19
    k = 64*[0]
    k = [
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1,
        0x923f82a4, 0xab1c5ed5,
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe,
        0x9bdc06a7, 0xc19bf174,
        0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa,
        0x5cb0a9dc, 0x76f988da,
        0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
        0x06ca6351, 0x14292967,

```

```
0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb,
0x81c2c92e, 0x92722c85,
0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624,
0xf40e3585, 0x106aa070,
0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
0x5b9cca4f, 0x682e6ff3,
0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb,
0xbef9a3f7, 0xc67178f2]

def ror(x, y):
    return ((x >> y) | (x << (32-y))) & 0xFFFFFFFF

for n in range(len(data)):
    bytes+='{0:08b}'.format(ord(data[n]))
    bits = bytes+"1"
    pBits = bits
    #pad until length equals 448 mod 512
    while len(pBits)%512 != 448:
        pBits+="0"
    #append the original length
    pBits+='{0:064b}'.format(len(bits)-1)

def chunks(l, n):
    return [l[i:i+n] for i in range(0, len(l), n)]

def rol(n, b):
    return ((n << b) | (n >> (32 - b))) & 0xffffffff

for c in chunks(pBits, 512):
    words = chunks(c, 32)
    w = [0]*64
    for n in range(0, 16):
        w[n] = int(words[n], 2)
    for i in range(16, 64):
        s0 = ror(w[i-15], 7) ^ ror(w[i-15], 18) ^ (w[i-15] >> 3)
        s1 = ror(w[i-2], 17) ^ ror(w[i-2], 19) ^ (w[i-2] >> 10)
        w[i] = (w[i-16] + s0 + w[i-7] + s1) & 0xffffffff

    a = h0
    b = h1
    c = h2
    d = h3
    e = h4
    f = h5
    g = h6
    h = h7

    for i in range(0, 64):
        S1 = ror(e, 6) ^ ror(e, 11) ^ ror(e, 25)
        ch = (e & f) ^ ((~e) & g)
        temp1 = h + S1 + ch + k[i] + w[i]
```

```

S0 = ror(a, 2) ^ ror(a, 13) ^ ror(a, 22)
maj = (a & b) ^ (a & c) ^ (b & c)
temp2 = S0 + maj

h = g
g = f
f = e
e = (d + temp1) & 0xffffffff
d = c
c = b
b = a
a = (temp1 + temp2) & 0xffffffff

h0 = h0 + a & 0xffffffff
h1 = h1 + b & 0xffffffff
h2 = h2 + c & 0xffffffff
h3 = h3 + d & 0xffffffff
h4 = h4 + e & 0xffffffff
h5 = h5 + f & 0xffffffff
h6 = h6 + g & 0xffffffff
h7 = h7 + h & 0xffffffff

return '%08x%08x%08x%08x%08x%08x%08x' % (h0, h1, h2, h3, h4, h5, h6, h7)

def is_bip66(sig):
    sig = bytearray.fromhex(sig)
    print "sig type %s" % type(sig)
    if sig[1] == len(sig)-2:
        sig.extend(b"\1") # add SIGHASH for BIP66 check
    if len(sig) < 9 or len(sig) > 73: return False
    if (sig[0] != 0x30): return False
    if (sig[1] != len(sig)-3): return False
    rlen = sig[3]
    if (5+rlen >= len(sig)): return False
    slen = sig[5+rlen]
    if (rlen + slen + 7 != len(sig)): return False
    if (sig[2] != 0x02): return False
    if (rlen == 0): return False
    if (sig[4] & 0x80): return False
    if (rlen > 1 and (sig[4] == 0) and not (sig[5] & 0x80)): return False
    if (sig[4+rlen] != 0x02): return False
    if (slen == 0): return False
    if (sig[rlen+6] & 0x80): return False
    if (slen > 1 and (sig[6+rlen] == 0) and not (sig[7+rlen] & 0x80)): return False
    return True

def ECadd(x1 , y1 ,x2 , y2):
    m = ( (y2 - y1 ) * inv1( x2 - x1 , Pcurve)) % Pcurve
    x3 = ( m * m - x1 - x2 ) % Pcurve
    y3 = ( m * ( x1 - x3 ) - y1 ) % Pcurve

```

```
    return (x3, y3)

def ECdouble(x1 , y1):
    m = ( (3 * x1 * x1 ) * inv1( (2 * y1 ), Pcurve)) % Pcurve
    x3 = (m * m - 2 * x1) % Pcurve
    y3 = (m * ( x1 - x3) - y1) % Pcurve
    return (x3,y3)

def EccMultiply(Gx , Gy , pkey):
    keybin = bin(pkey)[2:]
    x3 , y3 = Gx , Gy
    for i in range (1, len(keybin)):
        x3 , y3 = ECdouble(x3 , y3);
        if keybin[i] == "1":
            x3 , y3 = ECadd(x3 , y3 , Gx , Gy);
    return (x3 , y3)

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def decode58(string):
    code_string = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
    result = 0
    while len(string) > 0:
        result = result * 58
        result = result + code_string.find(string[0])
        string = string[1:]
    return result

def inv1(a, n):
    if a == 0:
        return 0
    lm, hm = 1, 0
    low, high = a % n, n
    while low > 1:
        r = high // low
        nm, new = hm-lm*r, high-low*r
        lm, low, hm, high = nm, new, lm, low
    return lm % n

def abc(ind):
    o = ""
    o = o + "01000000"
    o = o + "01"
    inp = thash
    o = o + reversehash(inp)
    outis = "%08x" % outi
    o = o + reversehash(outis)
    if ind == 0:
        scrLen = len(addr1)/2
```

```

    scripen = "%02x" % scripen
    o = o + scripen
    o = o + addr1
    else:
        scripen1 = len(dummy3)/2
        scripen2 = "%02x" % scripen1
        o = o + scripen2
        o = o + dummy3
        o = o + "ffffffff"
        o = o + "01"
        zz = "%016x" % bitcoins
        zz = reversehash(zz)
        o = o + zz
        scripen1 = len(addr2)/2
        scripen1 = "%02x" % scripen1
        o = o + scripen1
        o = o + addr2
        o = o + "00000000"
    return o

def privkey_to_pubkey1(priv):
    zz = decode58(priv)
    xx = "%x" % zz
    data = xx.decode('hex')
    data = data[1:-4]
    data = data[:32]
    privkey = int(data.encode('hex'), 16)
    if privkey >= N:
        raise Exception("Invalid privkey")
    multiply = EccMultiply(Gx, Gy, privkey)
    byte = str(2+(multiply[1] % 2))
    bb = "%x" % multiply[0]
    pub = '0' + byte + bb
    return pub

def mk_pubkey_script1(address):
    aa = decode58(address)
    bb = "%x" % aa
    bb = bb.decode('hex')
    zz = bb[:-4]
    aa = zz.encode('hex')
    script = '76a914' + aa + '88ac'
    return script

def make_request1(*args, **kwargs):
    nonce = random.randrange(999999)
    headers = kwargs.get('headers') or \
    { "User-agent": "Mozilla/5.0%d" % nonce,
      "Accept": "application/json",

```

```
}
url = args[0]
req = urllib2.Request(url, data=None, headers=headers)
return urllib2.urlopen(req).read().strip()

def history1(addr):
    txs = []
    offset = 0
    while 1:
        data = make_request1('https://blockchain.info/address/%s?format=json&offset=%s' % (addr, offset))
        #print data
        jsonobj = json.loads(data)
        txs.extend(jsonobj["txs"])
        if len(jsonobj["txs"]) < 50:
            break
        offset += 50
    outs = {}
    for tx in txs:
        for o in tx["out"]:
            try:
                if o.get('addr', None) in addr:
                    key = str(tx["tx_index"])+':'+str(o["n"])
                    outs[key] = {
                        "address": o["addr"],
                        "value": o["value"],
                        "output": tx["hash"]+':'+str(o["n"]),
                        "block_height": tx.get("block_height", None)
                    }
            except:
                pass
    for tx in txs:
        for i, inp in enumerate(tx["inputs"]):
            if "prev_out" in inp:
                if inp["prev_out"].get("addr", None) in addr:
                    key = str(inp["prev_out"]["tx_index"]) + \
                        ':' + str(inp["prev_out"]["n"])
                    if outs.get(key):
                        outs[key]["spend"] = tx["hash"] + ':' + str(i)
    #print [outs[k] for k in outs]
    return [outs[k] for k in outs]

def deterministic_generate_k1(priv, msghash):
    import hmac
    import hashlib
    dig = hmac.new(priv, msg=msghash, digestmod=hashlib.sha256).digest()
    dig1 = int(dig.encode('hex'), 16)
    return dig1
```



```

def encode58(val):
    code_string = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
    string = ""
    while val > 0:
        string = code_string[val % 58] + string
        val = val // 58
    return string

def lpad1(msg, symbol, length):
    if len(msg) >= length:
        return msg
    return symbol * (length - len(msg)) + msg

def pubkey_to_address1(pubkey):
    zz = pubkey.decode('hex')
    intermed = sha256(zz).decode('hex')
    length = ( len(intermed) << 3) & (2**64-1)
    data = intermed + "\x80"
    data = struct.pack("<56sQ", data, length)
    digest = ripemd160(data)
    digest = digest.decode('hex')
    inp_fmt = '\x00' + digest
    yy1 = sha256(inp_fmt)
    yy2 = sha256(yy1).decode('hex')
    yy = yy2.decode('hex')
    checksum = yy[:4]
    leadingzbytes = 0
    for x in inp_fmt:
        if x != 0:
            break
    leadingzbytes += 1
    nblen = len(re.match(b'^(\0)*', inp_fmt+checksum).group(0))
    yy = int((inp_fmt+checksum).encode('hex'), 16)
    yy = lpad1('', '1', nblen) + encode58(yy)
    return '1' * leadingzbytes + yy

def sendversionpacket(sock):
    magicbytes = "f9beb4d9"
    command = "version\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')

    #str1 = '62ea' + '0' * 166
    str1 = '7f110100' + '0' * 162
    checksumhash1 = chash(str1.decode('hex'))
    checksumhash1 = checksumhash1[:4]

    packetlength = len(str1)/2
    packetlengthstr = "%08x" % packetlength
    str = magicbytes + commanddecoded + reversehash(packetlengthstr) + checksumhash1.encode('hex')
    str2 = str + str1

```

```
    str2 = str2.decode('hex')
    sock.sendall(str2)

def chash(s):
    tem1 = sha256(s)
    final = sha256(tem1.decode('hex'))
    return final.decode('hex')

def decodeversion(payloadbytes , sock):
    magicbytes = "f9beb4d9"
    command = "verack\x00\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % 0
    checksum = chash(payloadbytes)
    checksumhash = checksum[:4]
    checksumhash = checksumhash.encode('hex')
    str = magicbytes + commanddecoded + length + checksumhash
    str = str.decode('hex')
    sock.sendall(str)

dictionary = {0:"Error" , 1:"Transaction" , 2:"Block" , 3:"Filtered Block"}

def decodegetdata(payloadbytes , sock):
    (count , ) = struct.unpack("c" , payloadbytes[:1])
    count = ord(count)
    payloadbytes = payloadbytes[1:]
    if count == 253:
        (count, ) = struct.unpack("h" , payloadbytes[: 2])
        payloadbytes = payloadbytes[2:]
        for i in range ( 0 , count ):
            (type , hash , ) = struct.unpack("I32s" , payloadbytes[i * 36 : i*36 + 36])
            magicbytes = "f9beb4d9"
            command = "tx\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
            commanddecoded = command.encode('hex')
            tbytes = rawoutput
            tbytes = tbytes.decode('hex')
            length = "%08x" % len(tbytes)
            #print "Length of raw bytes is " , length
            lengthrev = reverseahash(length)
            checksum = chash(tbytes)
            checksumhash = checksum[:4]
            checksumhash = checksumhash.encode('hex')
            str = magicbytes + commanddecoded + lengthrev + checksumhash
            str = str.decode('hex')
            str2 = str + tbytes
            print "Actually sending a transaction"
            sock.sendall(str2)
            time.sleep(4)
            exit(0)
```

```

def sendainvcommand(sock):
    magicbytes = "f9beb4d9"
    command = "inv\x00\x00\x00\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % 37
    lengthrev = reverseahash(length)
    count = '01'
    invtype = '01000000'
    reversethash = thash
    str1 = count + invtype + reversethash
    checksum = chash(str1.decode('hex'))
    checksumhash = checksum[:4]
    checksumhash = checksumhash.encode('hex')
    str = magicbytes + commanddecoded + lengthrev + checksumhash
    str2 = str + str1
    sock.sendall(str2.decode('hex'))

def decoding(payloadbytes, sock):
    if len(payloadbytes) == 8:
        (nonce,) = struct.unpack("Q", payloadbytes[:8])
        nonce = "%x" % nonce
    else:
        nonce = "0000"
    nonce = reverseahash(nonce)
    checksum = chash(nonce.encode('hex'))
    checksumhash = checksum[:4]
    checksumhash = checksumhash.encode('hex')
    magicbytes = "f9beb4d9"
    command = "pong\x00\x00\x00\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')
    length = "%08x" % 8
    lengthrev = reverseahash(length)
    noncelen = len(nonce)
    diff = 16 - noncelen
    str = magicbytes + commanddecoded + lengthrev + checksumhash + nonce + '0' * diff
    str = str.decode('hex')
    sock.sendall(str)
    #print "Sending a inv packet"
    sendainvcommand(sock)

def handlecommand(command, length, checksum, payload, cnt, sock):
    if "version" in command:
        decodeversion(payload, sock)
    elif "ping" in command:
        decoding(payload, sock)
    elif "getdata" in command:
        decodegetdata(payload, sock)

```

```
def sendtransaction():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    #sock.connect(("bitcoin.sipa.be", 8333))
    sock.connect(("127.0.0.1", 8333))
    sendversionpacket(sock)
    bytesrecieved = ""
    cnt = 0
    while (True):
        bytesrecieved = bytesrecieved + sock.recv(64 * 1024)
        (magicnumber , command, payloadlength , checksum ) = struct.unpack("I12sII" ,bytesrecieved[:24])
        #print "Command%s payloadlength %d" % (command , payloadlength)
        while len(bytesrecieved) >= payloadlength + 24:
            checksumstr = "%08x" % checksum
            payload = bytesrecieved[24:24 + payloadlength]
            cnt = cnt + 1
            handlecommand(command , payloadlength , reversehash(checksumstr) , payload , cnt , sock)
            bytesrecieved = bytesrecieved[24 + payloadlength:]
            if len(bytesrecieved) >= 24:
                (magicnumber , command, payloadlength , checksum ) = struct.unpack("I12sII" ,bytesrecieved[:24])

pub = privkey_to_pubkey1(priv)
print "pub=%s" % pub
address = pubkey_to_address1(pub)
print "source bitcoin address=%s" % address
addr1 = mk_pubkey_script1(address)
unspend = history1(address)
unspendl = []
for obj in unspend:
    if 'spend' not in obj.keys():
        unspendl.append(obj)
if len(unspendl) == 0:
    print "No balances to spend"
    exit(0)
spendd = unspendl[0]
print "Total Number are %d" % len(unspend)
print "Number of unsent outputs are %d" % len(unspendl)
print "spendd=%s" % spendd
newobject = spendd['output']
ind = newobject.index(':')
thash = newobject[0:ind]
outi = int(newobject[ind + 1:])
addr2 = mk_pubkey_script1(daddr)
signing_tx = abc(0)
hashstring = signing_tx.decode('hex') + "\x01\x00\x00\x00"
tmp2 = sha256(hashstring)
dummy = sha256(tmp2.decode('hex'))
dummy = dummy.decode('hex')
```

```

z = int(dummy.encode('hex') , 16)
k = deterministic_generate_k1(dummy, priv)
print "Random Number k=%s" % k
r, y = EccMultiply(Gx, Gy, k)
dummy10 = decode58(priv)
dummy10 = ("%x" % dummy10).decode('hex')
dummy1 = dummy10[1:-4][:32]
decodeprivkey = int(dummy1.encode('hex'),16)
s = inv1(k, N) * (z + r * decodeprivkey) % N
print "s=%s" % s
s = N - s if s > N // 2 else s # BIP62 low s
print "s=%s" % s
r1 = ("%064x" % r).decode('hex')
s1 = ("%064x" % s).decode('hex')
print "r1=%s" % r1.encode('hex')
print "s1=%s" % s1.encode('hex')
if bytearray(r1)[0] & 0x80:
    print "b1 negative"
    r1 = b'\x00' + r1
if bytearray(s1)[0] & 0x80:
    print "b2 negative"
    s1 = b'\x00' + s1
lenr1 = ("%02x" % len(r1)).decode('hex')
lens1 = ("%02x" % len(s1)).decode('hex')
print "lenr1=%s" % lenr1.encode('hex') , "lens1=%x" % int(lens1.encode('hex'),16)
left = b'\x02' + lenr1 + r1
print "left=%s" % left.encode('hex')
right = b'\x02' + lens1 + s1
print "right=%s" % right.encode('hex')
totallength = ("%02x" % len(left+right)).decode('hex')
print "Total length %d:%x" % (len(left+right),len(left+right))
sighex = (b'\x30' + totallength + left + right).encode('hex')
print "sighex=%s" % sighex
sig = sighex + '01'
print "sig=%s" % sig
assert is_bip66(sig)
lensig = len(sig)/2
lenpub = len(pub)/2
print "lensig=%0x" % lensig
print "lenpub=%0x" % lenpub
dummy3 = ("%02x" % lensig + sig + "%02x" % lenpub + pub
rawoutput = abc(1)
temp = sha256(rawoutput.decode('hex'))
final = sha256(temp.decode('hex'))
thash = reverseahash(final)
print "Final Transaction hash is %s" % thash
#os.system("bitcoin-cli sendrawtransaction " + rawoutput)

```

sendatransaction()

A transaction sent in July 2016

```
pub=038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3
source bitcoin address=1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n
Total Number are 18
Number of unsent outputs are 4
spendd={'output':
u'7c9d0463f507ae079dc6f0b6c35a2c734d01e526053416919e27854eee320893:0',
'block_height': 419356, 'value': 64100, 'address':
u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}
Random Number
k=686559497870354034202705347549373660312393404991436308919581146708730
29172518
s=6368436533627451154131896655721004510581649050376657979733419481803611
4279504
s=5210772390104168388225201845147786274702107377530832458527096832348204
7214833
r1=4631e302f56dd3cf8ec35637541b7efb67928b1e337d9ac0566ffb8a3f3465e1
s1=7333ed33ebb038eecddeebca49c4d3f53f4d34f0a7fe115bebf01f90822ff4f1
lenr1=20 lens1=20
left =02204631e302f56dd3cf8ec35637541b7efb67928b1e337d9ac0566ffb8a3f3465e1
right=02207333ed33ebb038eecddeebca49c4d3f53f4d34f0a7fe115bebf01f90822ff4f1
Total length 68:44
sighex=304402204631e302f56dd3cf8ec35637541b7efb67928b1e337d9ac0566ffb8a3f
3465e102207333ed33ebb038eecddeebca49c4d3f53f4d34f0a7fe115bebf01f90822ff4f1
sig=304402204631e302f56dd3cf8ec35637541b7efb67928b1e337d9ac0566ffb8a3f3465
e102207333ed33ebb038eecddeebca49c4d3f53f4d34f0a7fe115bebf01f90822ff4f101
sig type <type 'bytearray'>
lensig=47
lenpub=21
Final Transaction hash is 51d10c5fe13ceae5aaaccd84f40ae95d37acf2c5c380dc959d829eae1c48e47a
```

A transaction sent in the year January 2017

```
pub=038bad033170ac512b2c68e58cd93e929e9a92fe4e7bc36d0fe0b0fd365774dce3
source bitcoin address=1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n
Total Number are 34
Number of unsent outputs are 3
spendd={'output':
u'cc19400543100da4ae3ee048a079e03158b16d1d9a855a6abb7a67334c1a6395:1',
'block_height': 429572, 'value': 74600, 'address':
u'1Lg57gLSp8HYg6W41kwddNXLik6oUZ3p3n'}
Random Number
k=110608506118775854620561194441920558745853843530931515662598451299795
140274352
s=9956293646455740982358956008170096675452129060901979371007561623440603
056290
```

```

s=9956293646455740982358956008170096675452129060901979371007561623440603
056290
r1=1440a563b9794a0f9fc8baf56e4e652125f45e1b4c2ea83188497c6ba27475f5
s1=1603100050992d46465c772806b5faf0aa657dcffce4a5f6240e74029b4988a2
lenr1=20 lens1=20
left =02201440a563b9794a0f9fc8baf56e4e652125f45e1b4c2ea83188497c6ba27475f5
right=02201603100050992d46465c772806b5faf0aa657dcffce4a5f6240e74029b4988a2
Total length 68:44
sighex=304402201440a563b9794a0f9fc8baf56e4e652125f45e1b4c2ea83188497c6ba2
7475f502201603100050992d46465c772806b5faf0aa657dcffce4a5f6240e74029b4988a2
2
sig=304402201440a563b9794a0f9fc8baf56e4e652125f45e1b4c2ea83188497c6ba2747
5f502201603100050992d46465c772806b5faf0aa657dcffce4a5f6240e74029b4988a201
sig type <type 'bytearray'>
lensig=47
lenpub=21
Final Transaction hash is ecd0069ba004ca4246c1e9bcd24e81223d89fc50fe322cfd54ce4f9a7e41a022
Actually, sending a transaction

```

We have shown multiple outputs basically to identify the differences between transactions.

The code is tried using the command `sendrawtransaction` also and the transaction received is the following:

```
cbb527468f705529bd89a3b6b9809098cba13d72663f99fee725d27636f0d62a
```

We will send one last final transaction hash value :

```
7848069731f7dbf23700fdb26e9cb6fa512e1463dc3ebbb7c2890442da44ff18.
```

We are proud of ourselves as we have written most of the code in this program, apart from using the standard Python libraries. No code from Bitcoin Core is used so, you can stop the bitcoin application (but change the host localhost) and this code will still work as advertised. We call this clean code, everything in our control. We actually celebrated and wanted to stop writing this book but did not.

In October 2017, we were stress testing all our code. We did not trust our networking code so we used the method `sendrawtransaction` to send out a transaction.

```

Final Transaction hash is
4121f1b0f9389503c785b9d0eec3adc4196a6dcfbaeea01681c8eb37c2b18648
4121f1b0f9389503c785b9d0eec3adc4196a6dcfbaeea01681c8eb37c2b18648

```

These are the last two lines of the output. We get no error but this transaction hash is not recognized by any blockchain browser. We cannot figure out why we get no errors but the transaction hash is not visible anywhere. We would love to know where we have gone wrong.

So, this is what we did. We shut down the Bitcoin server, ran the same code again, no luck but we get the same transaction hash. We have to wait at the `sendrawtransaction` method. We then did something very scientific, we shut down our iMac and restarted. Now things work magically, and any blockchain explorer will confirm the date of the above transaction hash.

Once again if the code does not work, do not look at us. If you run the same code today, the message you will see is: No balances to spend.

CHAPTER 23

Index Folder

For some reason, nobody puts up code explaining the internals of Bitcoin like the index folder within the blocks folder, the format of the multiple rev files or the all-important UTXO's in the chainstate folder.

Let's understand the problem first. You want to store half a million records, which is too many to be contained in a simple plain text ASCII file. In such a situation, choosing the database option is ideal. But imagine people having to install a database server like SQL Server first and then install the Bitcoin core.

Fortunately, in the technology world, there are many embedded databases available that can store more than a million records. For reasons best known to them, the Bitcoin developers opted for two different embedded databases. An embedded database does not require any installation of the external program. It carries its code along with it and becomes a part of a larger program. An embedded database is embedded in a library.

First, we understand Leveldb, an embedded database written by the techies at Google. This database is open source. Then we will move to California and look at the BerkleyDB database.

Let's first install the leveldb bindings for python.

Command prompt \$: sudo pip install leveldb

Creating a Leveldb Folder

```
ch2301.py
import leveldb
db = leveldb.LevelDB("/Users/vijaymukhi/vijay123")
```

Output

```
[~]$ ls -l vijay123
total 24
-rw-r--r-- 1 vijaymukhi staff 0 Jul 7 11:49 000003.log
-rw-r--r-- 1 vijaymukhi staff 16 Jul 7 11:49 CURRENT
-rw-r--r-- 1 vijaymukhi staff 0 Jul 7 11:49 LOCK
-rw-r--r-- 1 vijaymukhi staff 57 Jul 7 11:49 LOG
-rw-r--r-- 1 vijaymukhi staff 50 Jul 7 11:49 MANIFEST-000002
```

To create a leveldb database, the leveldb function is called with one parameter. This parameter is given a folder name (along with full path) and the database gets installed there. Please note, a valid folder is specified and not a file. Our folder name is vijay123, you choose yours. Using shortcuts like ~ may not work.

Some files get created in this folder. All the records are present in these files.

Adding and Reading Keys to a Leveldb Database

```
ch2302.py
import leveldb
db = leveldb.LevelDB("/Users/vijaymukhi/vijay123")
db.Put("name", "Vijay")
print db.Get("name")
print db.Get("Name")
```

Output

Vijay

Traceback (most recent call last):

File "dummy.py", line 5, in <module>

print db.Get("Name")

KeyError

The Leveldb programming at one level is very simple. A database or a folder is created only once. If the folder is not present, it gets created. If the folder already exists, no error is reported.

Then data is added in key-value pairs in this database. The Put function creates one such key-value pair. The key is name and its value is my name, Vijay. There are really no rules on naming a key. The Get function retrieves the value associated with a key. Simply specify the key name and the value associated with this key is obtained. The keys names are case sensitive so if you give a wrong key name, a KeyError exception is thrown.

Iterating Through All the Keys in a Leveldb Database

```
ch2303.py
import leveldb
db = leveldb.LevelDB("/Users/vijaymukhi/vijay123")
db.Put("name", "Vijay")
db.Put("age", "10")
db.Put("salary", "30")
for k, v in db.RangeIter():
    print k, "—>", v
```

Output

age —> 10

name —> Vijay

salary —> 30

A leveldb database can store millions of key-value pairs. In this example, there are three keys; name, age and salary. The values assigned to them must be of string type and not say numbers. Python makes the transition from strings to numbers very simple, so it is no cause of concern. These three key value pairs are put in the database.

In a loop, we iterate through these millions of key-value pairs. This is where the RangeIter function steps in. It returns one key-value tuple at a time. Each key value pair is stored in variables k and v. A Google search shows that the original code calls the key name variable, k and the value variable, v. Follow the leaders is our mantra.

It is very convenient to individually access every row of one key-value pair in a loop; reading a leveldb database could not have been any easier.

Counting the Number of Key-Value Pairs in a Leveldb Folder

```
ch2304.py
import leveldb
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")
cnt = 0
for k, v in db.RangeIter():
    cnt = cnt + 1
print "Total number of key value pairs %d" % cnt
```

Output

Total number of key value pairs 420079

We also get this message at another point in time when we ran the same program

Total number of key value pairs 197572584

Now for some very heavy lifting. We do not want to play around with the original data. So, please copy the index folder from the Bitcoins/blocks directory to the Desktop. But before that, make sure that the Bitcoin GUI / the bitcoin daemon, bitcoind is shut down. The folder name with the full pathname is given to the Leveldb function each time.

The program simply iterates through all the key-value pairs in the levels database and increases the counter variable, cnt by 1.

The total number of key-value pairs is half a million or 20 million and Leveldb database processes them in no time. There could be more than 200 million key-value pairs in our leveldb folder.

```
$bitcoin-cli getblockhash 293374
0000000000000000485620aa56d1a61c14f56423336772f3211f987fc390000
```

The block hash value of a randomly chosen block number, 293374 is displayed. Enter the block number in blockchain.info and you will see the same result.

Reading the Key of a Block Stored in the Index Folder

```
ch2305.py
import leveldb
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")
hash = '0000000000000000485620aa56d1a61c14f56423336772f3211f987fc390000'
hash = reversehash(hash)
hash = '62' + hash
print hash
hash = hash.decode('hex')
print db.Get(hash).encode('hex')
```

Output

```
62000039fc87f911322f7736332364f5141ca6d156aa2056480000000000000000
86a84090f27e1d1680008ae7c53eb9d70d02000000ed2223ddcbba95c7cce6cc31ce5ca
7463be8ec7720232acb000000000000000f9bf954db4429571d2b379e49c624c3c9d0d8
d9fcb91f8123f22e2558ba8dab5003e395399db0019146ae23f
```

The hash variable and the block hash have the same value. The good old reversehash function reverses the hash. The endianness curse if you recall.

Then, a small b or ASCII 98 or hex 62 is added at the start of the reversed hash. This hash becomes a valid key in the index leveldb folder.

This program summarizes that tacking a 0x62(b) to the start of a block hash value gives a valid leveldb key name. All block hashes have a leveldb key name. We are now getting someplace.

Now, to make sense of the value associated with this block hash key.

Reading the Encoded Data Stored along with a Block Key

ch2306.py

```
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        print "chData %x offset %d condition %d n=%d" % (chData , offset , chData & 0x80 == 128 , n )
        offset = offset + 1
        print "n=%x n << 7=%x chData & 0x7F=%x %s %x:%d " % (n, n << 7 , chData & 0x7F , bin(chData) ,
            (n << 7) | (chData & 0x7F) , (n << 7) | (chData & 0x7F))
        n = (n << 7) | (chData & 0x7F)
        print "n = %d:%x" % (n,n)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

(nVersion , offset) = base128("\x86\xa8\x40", 0)
print "nVersion %d" % (nVersion)
print "Offset is %d" % offset

a = 3 << 1
print a ,
a = 3 << 2
print a ,
a = 3 << 3
print a ,
a = 3 << 4
print a ,
a = 3 << 5
print a ,
a = 3 << 6
print a ,
a = 3 << 7
print a
print "%x" % (7 << 7)
```

Output

chData 86 offset 0 condition 1 n=0

```
n=0 n << 7=0 chData & 0x7F=6 0b10000110 6:6
n = 6:6
chData a8 offset 1 condition 1 n=7
n=7 n << 7=380 chData & 0x7F=28 0b10101000 3a8:936
n = 936:3a8
chData 40 offset 2 condition 0 n=937
n=3a9 n << 7=1d480 chData & 0x7F=40 0b1000000 1d4c0:120000
n = 120000:1d4c0
nVersion 120000
Offset is 3
6 12 24 48 96 192 384
380
```

All code picked up from the Bitcoin source is simply converted from C++ into the Python programming language. Along the way, the names of the variables used in the original code like chData are retained. Any bugs, blame it on the Bitcoin core team, though we have yet to come across one.

For a long time, we have used decimal, binary, hexadecimal, a base64 variant called base58. Bitcoin has unleashed upon the world a wide variety of custom formats. Now we present one more variant, the base128 encoding.

A function called base128 is created to convert some decoded bytes into the base128 format. This function is given three bytes. These are the first three bytes of the value associated with the block hash key. These hardcoded bytes have been hex-coded. The return value in variable n has the original bytes that were encoded with the base128 encoding scheme.

An infinite while loop is used since we have no idea as to how many bytes are used for decoding. The offset variable comes in the way of understanding base128 encoding. Its value starts at 0 as this value is passed as a parameter. This infinite while loop reads one byte at a time. The value of the variable offset is incremented by one each time in the loop. This facilitates decoding a number starting from any offset in the string. Once again, please ignore the offset variable only this time, it has nothing to do with base128 decoding.

Now let's move on to understanding base128 encoding.

Then one at a time, the chData variable receives each byte of the string of bytes, passed as a number. This is the beauty of the ord function. The variable offset is given on both sides of the slice operator. They cancel each other out. Therefore, it can be safely assumed that only one byte is read at a time owing to the +1 after the colon, but from an offset decided by the variable offset. This is a pun.

The first value of the chData variable is 86, then a8 and finally 40. The numbers are displayed in hex on purpose.

Look at the hex number, 0x86. The last or top 4 bits are 8 and in binary an 8 is written as 1000. The topmost bit is a 1. Ditto for the second byte a8. The top most bit is a 1. This is because a is 10 in decimal which in binary is 1010. But the top most bit for a 40 has the top most bit as a 0 since 4 in binary is 0100.

The first rule of base128 encoding is that it is a variable length encoding. Thus, it is difficult to figure out the number of bytes the original value was encoded with. It's not like a string which starts with the number of bytes.

Every byte that has its uppermost bit as 1, signifies that the next byte also contains part of the encoding. When the high bit of any byte is 0, it indicates that it is the last byte of the encoding and it's time to exit. This condition is checked by the last if statement. It checks whether the high bit or bit 7 of the variable chData is 1 or not. In other words, the high bit of variable chData is set or not. This is a more technical explanation. If the condition results in true, the value in variable n is increased by 1 and the loop continues.

If, however, the seventh bit is 0, then the value stored in the offset variable is returned along with *n*, the actual original value.

When there is a bitwise and (&) with a 0, the original bit is forcibly being set to 0 irrespective of its original value. A bitwise and with a 1, and the value remains the same. A repetition, 123 times and counting.

However, if the bitwise and is with 7F or 01111111, the last 7 bits are 1, so they do not get changed. As the first 7 bits are 1, the original 7 bits retain its earlier value. The top most bit is removed from the equation as we are bitwise anding with a 0.

One more explanation to bit shifting: when all the bits are moved 1 to the left, the 8th bit gets dropped off. In effect, the value is multiplied by 2. So, when there is a shift of 3 bits to the left, it is multiplying the value by 8 and so on. We assume that you have not read the cryptography chapters where we explained this in greater detail.

Let's understand the decoded value in the variable *n*. There are two processes applied to it, First, we are masking all the bits but the high bit and secondly, the bits are shifted by 7. Then these values are bitwise or'ed.

The original value to be encoded was 120000 and the base128 encoding results in a value of 86a840. There is no change in size of the encoding. So, how does the 86a840 give the version number?

This is no chicken-egg situation as to who comes first. In any encoding, first the value is encoded and when it is decoded, the original value is obtained. The Bitcoin world created the base128 encoding as a variable length encoding, so that no one has any prior knowledge as to how many bytes are to be decoded to get back the original value.

The first byte in variable *chData* is 0x86. The task now is to determine the encoded value of the value in the *n* variable because on decoding, this variable will contain the original value. The variable *n* is made up of two different computations. The left shifting by 7 gives a 0 as the value in *n* is 0. This is part A. Then, the *bin* function discloses the number of 1 bits in the *chData* byte.

The number 0x86 has 3 1 bits, the top bit is zeroed out. So now there are only 2 bits. Bits 1 and 2 are one and they have a weight of 2 and 4, thereby giving an answer of 6. This is part B.

Bitwise Oring A and B i.e. $0 | 6$, gives the value of 6 in variable *n*. The if statement is True so, 1 is added to the value of *n* thus giving a value of 7 at the start of the loop. So far so good.

In the second loop iteration, the value of variable *chData* is 0xa8. The *n* variable is left shifted by 7, thus giving a value of 380. This value is displayed on the screen. The value of Part B or *chData* & 0x7f is 0x28. The answer of *n* at this point is 0x3a8, close to the final value. At the end of the loop the variable *n* is increased by 1, thus resulting in a value of 3a9.

For the last time, the value in A and B is calculated which is 1d480 and 40. After all the bitwise operations, it finally gives an answer of 120000. The code is simply undoing the steps of encoding.

The offset value is finally three as there are three iterations in the for loop. The value of *n* is incremented by 1 each time in the loop. It ensures that the encoding is one-to-one.

The advantages of base128 encoding are that very small numbers, ranging from 0 to 127, will use just one byte. Numbers from 128-16511 takes up 2 bytes and numbers from 16512-2113663 will take up 3 bytes. If the normal rules are followed, each number would take up 4 bytes. There is also something called variable length encoding.

According to the comments in the Bitcoin code, the encoding does not depend on the size of the original integer type.

Decoding all the Fields Stored with the Block Key

```
ch2307.py
import leveldb
import struct
```

```
import time
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)
def index(ans):
    (nVersion , offset) = base128(ans , 0)
    print "nVersion %d" % (nVersion)
    (nHeight, offset) = base128(ans , offset)
    print "nHeight %d" % (nHeight)
    (nStatus, offset) = base128(ans, offset)
    print "nStatus %x:%d" % (nStatus , nStatus)
    (nTx, offset) = base128(ans, offset)
    print "nTx %d" % (nTx)
    (nFile, offset) = base128(ans, offset)
    print "nFile %d" % (nFile)
    (nDataPos, offset) = base128(ans, offset)
    print "nDataPos %d" % (nDataPos)
    (nUndoPos, offset) = base128(ans, offset)
    print "nUndoPos %d" % (nUndoPos)
    (version,prevblockhash , merklehash , creationtime , difficulty , nonce) =
    struct.unpack("I32s32sIII" , ans[offset : offset + 80 ])
    print "Version %d" % version
    print "PrevBlock Hash %s" % prevblockhash[:-1].encode('hex')
    print "Merkle Root %s" % merklehash[:-1].encode('hex')
    print time.ctime(creationtime)
    print "Difficulty %d" % difficulty
    print "Nonce %d" % nonce
    final = offset + 80
    print "Final byte %s" % final

db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index')
hash = '62000039fc87f911322f7736332364f5141ca6d156aa2056480000000000000000'
print "Reversed Hash:%s:%d" % (reversehash(hash) , len(hash))
hash = hash.decode('hex')
ans = db.Get(hash)
print "Key hash is :%s:%d" % (hash.encode('hex') , len(hash.encode('hex')))
print "%s:%d" % (ans.encode('hex') , len(ans))
index(ans)
```

Output

```

Reversed Hash:0000000000000000485620aa56d1a61c14f56423336772f3211f987fc39000062:66
Key hash is :62000039fc87f911322f7736332364f5141ca6d156aa2056480000000000000000:66
86a84090f27e1d1680008ae7c53eb9d70d02000000ed2223ddcbba95c7cce6cc31ce5ca
7463be8ec7720232acb000000000000000f9bf954db4429571d2b379e49c624c3c9d0d8
d9fcb91f8123f22e2558ba8dab5003e395399db0019146ae23f:97
nVersion 120000
nHeight 293374
nStatus 1d:29
nTx 22
nFile 128
nDataPos 24781630
nUndoPos 961549
Version 2
PrevBlock Hash 0000000000000000cb2a232077ece83b46a75cce31cce6ccc795bacbdd2322ed
Merkle Root b5daa88b55e2223f12f891cb9f8d0d9d3c4c629ce479b3d2719542b44d95bff9
Mon Mar 31 15:35:52 2014
Difficulty 419486617
Nonce 1071802900
Final byte 97

```

Now let's make sense of all the values stored with the block hash key in the index database. The index function displays these fields.

The first 7 fields are numbers encoded with the base128 encoding scheme.

Initially, the offset parameter of the function base128 has a value of 0. Since we are dealing with a variable length integer encoding, it will be difficult to determine the number of bytes used by the first field. Therefore, the offset returned is the position where the next byte should be read from. The index function is given the offset (the starting point) of the next integer.

The base128 function returns the start position of the next field. This point becomes the starting point to read the next field. Once again, this process is repeated, as every field can occupy as many bytes depending upon its value.

The base128 function is given an offset where it should be reading from. The return value indirectly gives the count of read bytes. Or in other words, from where it should be reading the next field.

Following the base128 numbers, is the same block header, that comes before the transaction data.

The first field is the version field. We see different values like 130100 or 120100. We have seen values like 110200 when running an older version.

The next field is called nHeight or the block number, everyone calls it the block height. Here, it is a random value we choose, 293374.

Then comes the field nFile which identifies the blk file where this block's data is found. A value of 128 means blk00128.dat. On our home machine, the value displayed is 127, as the blocks on different blockchains have no formal ordering.

The nDataPos field determines the position of the block in the file named blk00128.dat. In our case, this block is stored at the point 24781630 bytes from the start. It has the magic number for this block. Your mileage will obviously vary.

The undoPos deals with rev files which is the subject of a later chapter. The two fields play the same role but the blk file name now starts with rev instead.

Then comes the number of transactions in this block, tx. This will remain the same till the chickens come home to roost. Finally, there is a Status byte, it needs a program of its own.

The value stored in the offset variable is displayed at the end, it is 97. It is the number of bytes the base128 encoded numbers uses, 17, and the size of the block header 80. The length of the variable ans, i.e. len(ans) has the same value.

Every byte of the value associated with the key has been accounted for.

Let's now check if the block index data is correct or not.

First, create the following table in PostgreSQL.

```
create table blockdata (fno integer , blockno integer , filepos integer , blockhash varchar(64) ,
prevhash varchar(64) , merkle varchar(64) , times varchar(256) , diff bigint , nonce bigint ,
version integer , txno integer);
```

Adding Records to the Blockdata Table

```
ch2308.py
from cfuncs import *
import psycpg2
import time
conn = psycpg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
def rvarint1(f):
    size = ord(f.read(1))
    if size < 0xfd :
        f.seek(-1,1)
        return size
    if size == 0xfd:
        zz = struct.unpack('H' , f.read(2))[0]
        f.seek(-3,1)
        return zz
    return -1
blockno = 0
for fno in range(0 , 1000):
    fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat" % fno
    try:
        f = open(fname , "rb")
        print "Opened File %s:%d" % (fname,blockno)
    except:
        break
    while ( True ):
        try:
            where = f.tell()
            magic = rint(f)
            if magic == 0:
                continue
        except:
```



```

break
size = rint(f)
header = f.read(80)
blockhash = chash(header)
(ver , prevhash , merklehash , time1 , diff , nonce ) =
struct.unpack("l32s32slll" , header)
txno = rvarint1(f)
s1 = "insert into blockdata values (%d , %d , %d , '%s' , '%s' , '%s' , '%s' , %d , %d , %d , %d)"
% (fno , blockno , where + 8 , blockhash.encode('hex') , prevhash[:-1].encode('hex') ,
merklehash[:-1].encode('hex') , time.ctime(time1) , diff , nonce , ver , txno)
#print s1
cur.execute(s1)
blockno = blockno + 1
f.seek(size - 80, 1)
conn.commit()

```

Output

```

select count(*) from blockdata;
count
-----

```

```

485483
(1 row)

```

Adding block data to a table

```

postgres=# select * from blockdata where blockhash =
'000039fc87f911322f7736332364f5141ca6d156aa2056480000000000000000';
fno | blockno | filepos | blockhash | prevhash | merkle | times | diff | nonce | version | txno
128 | 293458 | 24781630 | 000039fc87f911322f7736332364f5141ca6d156aa2056480000000000000000 |
00000000000000000000cb2a232077ece83b46a75cce31cce6ccc795bacbdd2322ed | b5daa88b55e2223f12f891c
b9f8d0d9d3c4c629ce479b3d2719542b44d95bfff9 | Mon Mar 31 15:35:52 2014 | 419486617 | 1071802900
| 2 | 22
(1 row)

```

This program reads all the blocks in every blk.dat file. It adds one record in the blockdata table for every block it encounters. Basically, the data is similar to the one present in the block header of the block hash plus more.

Let's begin with the file number of the block being read. It is present in the variable fno. It is followed by the height or block number, stored in the variable blockno. The start position of the block header in the file is stored in variable where. We add 8 to account for the space taken up the magic number and the file size.

The value stored in the index database is basically the starting point of the block header up to the start of the block.

Then, the encoded values of the blockhash is stored, it is the hash value calculated using the chash function. The merkle hash and previous block hash values are read for the block header. The values are reversed before writing the values to the database.

Thereafter comes the time of block creation in string form, the difficulty and nonce as integers. The version number of the block stored in the block header is also taken but the count/number of transactions in the block is however stored outside the block header.

The function `rvarint1` behaves a little differently. The first byte of the integer variable is read; it stores the number of transactions. We have moved one byte ahead. If the number of transactions is below `0xfd`, then the size bytes occupies just one byte. The file pointer is moved back by 1 and the size is returned. If, however, the first byte read is `0xfd`, the next two bytes are read. The `unpack` function performs its tasks and returns a short value, this value is the final count of transactions in the block. Since three bytes are read, the file pointer is moved back three bytes. It is assumed that no transaction will have more than 65000 plus transactions.

At the end of the loop, the pointer is at the start of the next block. The function `rvarint1` assumes that nothing is read and it positions the file pointer to where it originally was.

Now that about half a million blocks of data are stored in the block data table, it's time to query the table for a certain block hash value. The block hash key is always prefaced by a `b` or `0x62`. The outputs will differ here. More so, because the block number will not match as blocks on disk are not stored in an organized way. However, the number of transactions in the block, the file number, the data position, transaction version and the other block header fields match to a `T`.

Once again, the file position and file number may not match, the date time field will match.

Different Types of Keys and their Frequencies in the Index Folder

```
ch2309.py
import leveldb
cntb = 0
cntf = 0
cntF = 0
cntl = 0
cntR = 0
cntt = 0

db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")

for k,v in db.RangeIter():
    if k[:1] == 'b' :
        cntb = cntb + 1
        continue
    elif k[:1] == 'f' :
        cntf = cntf + 1
        continue
    elif k[:1] == 'F' :
        cntF = cntF + 1
        continue
    elif k[:1] == 'l' :
        cntl = cntl + 1
        continue
    elif k[:1] == 'R' :
        cntR = cntR + 1
        continue
    elif k[:1] == 't' :
        cntt = cntt + 1
        continue
```

```

else:
    print "Unknown type"
    print k[:1]
    print k.encode('hex')
    print v.encode('hex')
    exit(0)

print "cntb=%d" % cntb
print "cntf=%d" % cntf
print "cntF=%d" % cntF
print "cntl=%d" % cntl
print "cntR=%d" % cntR
print "cntt=%d" % cntt

```

Output

```

cntb=453132
cntf=774
cntF=1
cntl=1
cntR=0
cntt=196589280

```

This program highlights these six different types of keys in the Bitcoin index leveldb database, b, f, F, l and R and a t. If a new index type comes up, the print statements in the else will get called.

So, first in the for loop, every leveldb record is read and then the first byte of the key is checked. Depending on the type of the key, a variable is incremented by 1.

The three keys F, l and R will have the same values but the first two keys b and f will always be different than what we have shown you. The last one t will also be very different.

This summarizes the fact that the transaction hashes or type t occur very often, followed by block hashes or type b and then the f type which deals with every block file on disk. There are 565 different blk files on our hard disk and 419512 blocks in those files. Your mileage will vary depending upon the number of blocks you have downloaded. In October 2107, we have 1026 different block files and 261321081 transaction hashes and 489464 blocks.

If you do not see the type t, you are lucky.

Scanning all the Block Keys in the Index Folder

```

ch2310.py
import leveldb
import time
import struct
import psycopg2
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
cnt = 0
def base128(ans , offset):
    n = 0
    while True:

```

```
chData = ord(ans[offset:offset + 1])
offset = offset + 1
n = (n << 7) | (chData & 0x7F)
if chData & 0x80 == 128:
    n = n + 1
else:
    return (n,offset)
def indexcheck(ans , key):
    if key == "":
        print "Key is blank"
        exit(0)
    (nVersion , offset) = base128(ans , 0)
    (nHeight, offset) = base128(ans , offset)
    (nStatus, offset) = base128(ans, offset)
    (nTx, offset) = base128(ans, offset)
    (nFile, offset) = base128(ans, offset)
    (nDataPos, offset) = base128(ans, offset)
    (nUndoPos, offset) = base128(ans, offset)
    s1 = ""
    if len(ans) - offset == 80:
        (version,prevblockhash , merklehash , creationtime , difficulty , nonce) =
        struct.unpack("l32s32slll" , ans[offset : offset + 80 ])
        key = key.encode('hex')
        s1 = "Select * from blockdata where blockhash='%s'" % key
        cur.execute(s1)
        rows = cur.fetchone()
        if not (rows[4] == prevblockhash[::-1].encode('hex') and rows[5] == merklehash[::-1].encode('hex') and
        rows[6] == time.ctime(creationtime) and rows[7] == difficulty and rows[8] == nonce and rows[9] ==
        version and rows[10] == nTx):
            print "Error in key %s" % key
            print "rows[0] == nFile " , rows[0] == nFile , rows[0] , nFile
            print "rows[2] == nDataPos " , rows[2] == nDataPos , rows[2] , rows[2]
            print "rows[4] == prevblockhash[::-1].encode('hex') " , rows[4] == prevblockhash[::-1].encode('hex') ,
            rows[4] , prevblockhash[::-1].encode('hex')
            print "rows[5] == merklehash[::-1].encode('hex') " , rows[5] == merklehash[::-1].encode('hex') ,
            rows[5] , merklehash[::-1].encode('hex')
            print "rows[6] == time.ctime(creationtime) " , rows[6] == time.ctime(creationtime) , rows[6] ,
            time.ctime(creationtime)
            print "rows[7] == difficulty " , rows[7] == difficulty , rows[7] , difficulty
            print "rows[8] == nonce " , rows[8] == nonce , rows[8] , nonce
            print "rows[9] == version " , rows[9] == version , rows[9] , version
            print "rows[10] == nTx " , rows[10] == nTx , rows[10] , nTx
            print
            #exit(0)
        else:
            print "len ans %d len key %d offset %d diff %d" % (len(ans) , len(key) , offset , len(ans) - offset )
```

```

print "%s" % ans.encode('hex')
print type(key)
print key.encode('hex')
print s1

db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")
for k,v in db.RangeIter():
    if ord(k[:1]) == 0x62:
        indexcheck(v , k[1:])
        cnt = cnt + 1
    if cnt % 5000 == 0:
        print cnt , k.encode('hex') , "passed"

```

Output

```

len ans 95 len key 32 offset 16 diff 79
86a92498c3370b8243832dafb6e70f0100002010fa40986054f4f6592b19797d6771bd04
d568257bc7ba03000000000000000d7f824968467e6b5ea8a176bbdbadc0b75a24d6a0
9d74a1c30625c98b69679361e777257d63f0518252dd6a6
<type 'str'>
1179942a15b0bbea014858e220124a90c2625fd10708c9010000000000000000

```

Output

```

len ans 88 len key 32 offset 9 diff 79
86a924000b0100080100000000000000000000000000000000000000000000000000000000
00000000000000003ba3edfd7a7b12b27ac72c3e67768f617fc81bc3888a51323a9fb8aa4
b1e5e4a29ab5f49ffff001d1dac2b7c
<type 'str'>
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d619000000000000

len ans 88 len key 32 offset 11 diff 77
86a92498c670020001000020de8b35516f0a88cac4ce4b27639e6583a440b914793ebb0
20000000000000000032b632641f0e6cfd97bfbe30495c26eb88ec2af45ca51c710da11b6
bb75fe7002d377657d63f05183360a49a
<type 'str'>

len ans 95 len key 32 offset 16 diff 79
86a84098b8220b864e8324b8e3924801000020ab385ec1dd359b1c7f14c6d2219803b02
5fc1eb12eca0004000000000000000d6c9dae61629f9b2271dc1452ca8362a5c3f04a6b
c394e78780cf1716468f638d6d46557a09b0518460954fe
<type 'str'>

```

This program runs for hours on end but displays very little on the screen. In a for loop, every key-value pair is read in variables *k* and *v* respectively. With the *ord* function, the first byte of the key is checked to be 0x62 or a b. If so, the key represents a block hash value.

If the if statement is true, a modified *indexcheck* function is called. The function *indexcheck* is given two parameters. The first is the value stored with the block hash and the second is the block hash minus the first byte b.

In the *indexcheck* function, first we ensure that there is a valid block hash. There will never be a key that starts with a b and it is of length 1. The error check is for effect only. Then the first 7 fields from the value field are read as before. The variable *offset* now points to the start of the block header, this value is saved in the *ans* variable.

The block header is a constant 80 bytes in size. If the `ans` variable is not exactly 80 bytes long, there is a serious problem. This could happen if the block header is either truncated or there are extra bytes in the variable. Both are unacceptable. The `else` statement generally is never called.

The block header fields are read using the familiar unpack command.

A dynamic select statement is created which returns one row where the blockhash field of the blockdata table matches the key stored in the index leveldb database. The key has the block hash value. Thereafter, only the constant values across all copies of the downloaded blockchain are compared. It is futile to compare stuff like file numbers or block numbers that change from blockchain to blockchain.

The following fields are compared. The previous block hash, the merkle hash, the time, the difficulty, nonce, version and finally the number of transactions.

If any one of these is false, something is wrong somewhere so we simply display all the comparisons.

This program takes too much time to run only because the select must go through half a million records and there are no optimizations in our code. The other 300 million odd keys starting with t are ignored.

Our output shows only 4 blocks giving a size mismatch problem. An error of 4 in half a million can be ignored.

When the same code was executed in the year 2017, this is what we get

[illegible]

There is only one error now. Each time this program gives a different set of errors, but never exceeding four. We will not ask why these errors take place.

Decoding the nStatus Field of the Index Folder

```
ch2311.py
import leveldb
import struct
import time
offset = 0
def base128():
    global offset
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        if chData & 0x80 == 128:
            n = (n << 7) | (chData & 0x7F)
            n = n + 1
            offset = offset + 1
        else:
            n = (n << 7) | (chData & 0x7F)
            offset = offset + 1
    return n
```

```

def fstatus(nStatus):
    ans = ''
    s = bin(nStatus)
    if nStatus & 0x1 == 1:
        ans = ans + 'BLOCK_VALID_HEADER '
    if nStatus & 0x2 == 2:
        ans = ans + 'BLOCK_VALID_TREE '
    if nStatus & 0x4 == 4:
        ans = ans + 'BLOCK_VALID_CHAIN '
    if nStatus & 0x8 == 8:
        ans = ans + 'BLOCK_HAVE_DATA '
    if nStatus & 16 == 16:
        ans = ans + 'BLOCK_HAVE_UNDO '
    if nStatus & 32 == 32:
        ans = ans + 'BLOCK_FAILED_VALID '
    if nStatus & 64 == 64:
        ans = ans + 'BLOCK_FAILED_CHILD '
    return ans
db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index')
for k, ans in db.Rangelter():
    if ord(k[:1]) == 0x62:
        base128()
        base128()
        nStatus = base128()
        print "nStatus %x:%d" % ( nStatus , nStatus)
        print fstatus(nStatus)
        offset = 0
        break

```

Output

nStatus 1d:29

BLOCK_VALID_HEADER BLOCK_VALID_CHAIN BLOCK_HAVE_DATA BLOCK_HAVE_UNDO

The nStatus variable is a bit mask where every bit decides the status of the block. The various values stand for individual states.

In the function fstatus, there is a bitwise and to check if a bit in every position is on or off. If it is on, the check is displayed reconfirming the checks that block had passed.

It is assumed that the individual bits have single values; multiple bits are ignored for now. The nStatus field for the half a million key-value pairs is the same.

A value of 1 or BLOCK_VALID_HEDAEER means that the block passed the following checks. It has parsed for errors correctly, the version number is right, the hash satisfies the proof of work, the timestamp of the block creation date is not in the future.

A value of 2 or BLOCK_VALID_TREE implies that the parent block header is found, the difficulty matches, the same timestamp is larger than or equal to the medium timestamp of the previous block, checkpoint passes. All parents of every block are present right up to the genesis block.

A value of 3 is not a combination of the first 2, but implies more stringent error checks. For example, the first transaction in a block is always a Coinbase transaction. The length of the input script is less than 100, no checks on the output script length. This fairly explains the misuse of only outputs and not inputs.

The transactions are all valid in this block. There are no duplicate transactions id's. A check is performed for sigops, size and merkle roots.

A value of 4 or BLOCK_VALID_CHAIN implies that the sum of the outputs is not greater than inputs, one cannot spend more than what they own. No debt allowed. Plus, the same output cannot be spent twice, no double spends. The Coinbase output is okay but the input script has no meaning here.

The rules laid down in BIP30 are being observed.

A value of 8 or BLOCK_HAVE_DATA signifies that the block is available in a .dat file.

A value of 16 or BLOCK_HAVE_UNDO indicate that rev or undo files have the block representations present in them.

The last two checks are ignored.

We have simply tried to rewrite what was already given in the source code. You cannot question the above statements as they are visible only in comments in the source code.

An index key starting with f or file

```
ch2312.py
import leveldb
import struct
import time
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index')
i = 0
for k , v in db.RangeIter():
    if ord(k[:1]) == 0x66:
        i = i + 1
        if k[:1] == 'f':
            print k.encode('hex')
            block = struct.unpack("l" , k[1:])[0]
            print "Block Number %d" % block
            print v.encode('hex')
            (numberofblocks , offset) = base128(v , 0)
            (filesize , offset) = base128( v , offset)
            (revfilesize, offset) = base128(v , offset)
```



```

print "Number of blocks in the file is %d" % numberofblocks
print "Block File Size is %d" % (filesize - 8)
print "Rev Block Size is %d" % revfilesize
(lowestblocknumber, offset) = base128(v , offset)
(highestblocknumber, offset) = base128(v , offset)
(lowesttime, offset) = base128(v , offset)
(highesttime, offset) = base128(v , offset)
print "Lowest Block Number in file %d" % lowestblocknumber
print "Largest Block number in file is %d" % highestblocknumber
print "Smallest Time in Block is %s" % time.ctime(lowesttime)
print "Highest Time in Block is %s" % time.ctime(highesttime)
print
break

```

Output

```

f
6601020000
Block Number 513
8038beeff27f86fcf14f988608988c1284b8b19c4e84b8cafa30
Number of blocks in the file is 184
Block File Size is 133970423
Rev Block Size is 16742735
Lowest Block Number in file 410504
Largest Block number in file is 411282
Smallest Time in Block is Fri May 6 18:02:14 2016
Highest Time in Block is Wed May 11 15:08:56 2016

```

The second type of index key starts with a f or 0x66. The rest of the key is not a base128 number but a simple integer. The first byte is multiplied by 1 and the second byte by 256 and so on.

The key visually starts with a 0x66, the number 1 is multiplied by 1 and then, the next number 2 is multiplied with 256. The file number $(1+512) = 513$ is obtained in this manner.

The unpack function of the module struct does all the heavy lifting work. Our purpose is to display only one key beginning with f.

This number in the key is the file number. In our case, we choose the file number 513 to display the keys value in a readable form. The value starts with the number of blocks in the physical file with the file number which as we said before, is part of the key name. Then is the file size of the blk file and then the rev file details. All these fields are stored as base128 numbers.

The next four values cannot be checked as our physical blocks are not stored in any order. These are the starting and ending block numbers along with the smallest and largest date/time of the blocks in this physical file. The total number of blocks on our hard disk is over a 1000, but this number keeps growing over time.

Let's create the fileblock table in PostgreSQL.

```
create table fileblock(fileno integer , blockperfile integer , filesize bigint);
```

Storing the Physical File Data into a Database

```
ch2313.py
from cfuncs import *
import psycopg2
import os
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
for fno in range(0 , 1000):
    fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat" % fno
    try:
        f = open(fname , "rb")
        print "Opened File %s" % (fname)
    except:
        break
    blockspersfile = 0
    while ( True ):
        try:
            where = f.tell()
            magic = rint(f)
            if magic == 0:
                continue
        except:
            break
        size = rint(f)
        f.seek(size, 1)
        blockspersfile = blockspersfile + 1
        f.close()
        stat = os.stat(fname)
        fsize = stat.st_size
        s1 = "insert into fileblock values (%d , %d , %d )" % (fno , blockspersfile , fsize )
        cur.execute(s1)
    conn.commit()
```

Output

Opened File /Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk00000.dat

Opened File /Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk00001.dat

Running the code in the year 2016 or using a previous version of Bitcoin Core, we get the following output:

```
select * from fileblock where fileno = 513;
fileno | blockpersfile | filesize
-----+-----+-----
513 | 184 | 134093074
```

Running the same code in the year 2017 begin and end, using a newer version, gives this output.

```
select * from fileblock where fileno = 513;
fileno | blockperfile | filesize
-----+-----+-----
  513 |         184 | 133970423
(1 row)
```

This program simply fills up the fileblock table with the number of blocks found in each .dat file and its file size. At the end of the while loop, one block is read. And at the exit of the while loop, one physical dat file will be successfully read.

First, a unique physical file number is inserted. The blocks per file is obtained by simply incrementing the blockperfile variable by 1, each time in the while loop. The file size is obtained using the badly named stat function and its member st_size. Surprisingly to get the file size, it takes two lines of code and not one.

The select statement gives an output where the number of blocks match. It is file number 513. Please see the earlier output. But the file size was off by 8 bytes.

```
ls -l ~/Library/Application Support/Bitcoin/blocks/blk00513.dat
-rw---- 1 vijaymukhi staff 133970423 Jan 4 02:03
/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk00513.dat
```

Running the same code twice in the year 2017 showed the same file size. Anyways, the size is not important in our program.

Checking the Consistency of the f Keys

```
ch2314.py
import leveldb
import struct
import psycpg2
conn = psycpg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def indexcheck(v , block):
    (numberofblocks , offset) = base128(v , 0)
    (filesize , offset) = base128( v , offset)
    s1 = "Select * from fileblock where fileno=%d" % block
    cur.execute(s1)
    rows = cur.fetchone()
    if not (rows[1] == numberofblocks and (rows[2] == filesize or rows[2] == filesize -8)):
        print "Mismatch in File Number %d noblocks %d filesize %d" % (block, numberofblocks , filesize)
```

```
print "Mismatch in File Number %d noblocks %d filesize %d – Database" % (rows[0], rows[1], rows[2])
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")
for k,v in db.RangeIter():
    if ord(k[:1]) == 0x66:
        block = struct.unpack("l", k[1:])[0]
        indexcheck(v, block)
```

Output

```
Mismatch in File Number 562 noblocks 146 filesize 94678286
Mismatch in File Number 562 noblocks 194 filesize 133360756 – Database
```

The same code in the year 2017 gives the following output.

```
Mismatch in File Number 773 noblocks 21 filesize 20976358
Mismatch in File Number 773 noblocks 21 filesize 33554432 - Database
```

This program runs much faster as it has less than a 1000 keys to go through. It simply checks if the number of blocks in one dat file and its file size matches with the data stored in our database.

The first byte of the key *k* is checked to be 0x66 or a *f*. The function *indexcheck* is given two parameters. The first parameter is the entire value field. The second parameter is the file number that makes up for the rest of the key name. The *unpack* function gives this file number. Only the number of blocks and the file size is extracted from the value of the *f* key.

The *select* statement reads the row that matches the *fileno* field. The condition is that the number of blocks and the file size must be the same. But the file size sometimes may be off by 8. Some files have the same size, some different. Therefore, the *or* operator is used.

The block number count match totally.

We get an error when running our code on a live system. There is a constant download of our blocks and some blocks were left out after copying them to the index folder. Cannot call this an error. But it can be concluded that the blocks are saved in file *blk00773.dat*. The *.dat* files keep getting filled up all the time. The index was created some time ago and it is static, hence the second file size is much smaller.

That's why it's a good idea to add the following if condition as the index folder is not in sync compared to the files on disk.

If rows is not None:

This proves that the keys beginning with *f* pass the test.

A Key Called F

```
ch2315.py
import leveldb
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")
for k,v in db.RangeIter():
    if k[:1] == 'F' :
        print k.encode('hex')
        length = ord(k[1:2])
        print "length of string following %d" % length
```

```
print "Actual String %s" % k[2:length + 2]
print v
    break
```

Output

```
46077478696e646578
length of string following 7
Actual String txindex
1
```

The key that starts with F is the simplest key so far. The next byte gives the length of the string following, in our case it is only 7 bytes. We start from the third byte and display the string, which is txindex.

The value is a boolean so it will be 0 for false, 1 for true. In our case, the value is 1, so transaction indexing is enabled. When the value is 0, then transaction index is not enabled.

A Key Called I

```
ch2316.py
import leveldb
import struct
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")
for k,v in db.RangeIter():
    if k[:1] == 'I' :
        print k.encode('hex')
        print v.encode('hex')
        print struct.unpack("I" , v)[0]
        break
```

Output

```
6c
05030000
773
```

The key starting with I or 6c has a value that gives information on the last processed block file. It is an integer that can be unpacked. It has a value of 773; in October 2017 the value is 1025. Every time a block file is added in the blocks folder; this key I also changes its value. This identifies the file number where the blocks are currently being added.

Please confirm the output by checking the blocks folder and the last blk file number.

A Key Called R that Spells Danger

```
ch2317.py
import leveldb
#db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")
db = leveldb.LevelDB("/Users/vijaymukhi/Library/Application Support/bitcoin/blocks/index")
for k,v in db.RangeIter():
    if k[:1] == 'R' :
        print k.encode('hex')
        print v
```

Output

52

1

This program will not work on your machine. We sat on our heads to get data on a key beginning with a R. If this key's value is 1, it means that a re-indexing of the entire Bitcoin blocks is in process. It does not take months but less than a week to re-index the blocks as many files are created from scratch, like the index folder.

However, we modified the Bitcoin source in such a way that the re-indexing stopped after 172 blocks. Once you have the source, you can do anything you want.

As the re-indexing is incomplete or in process, this key starting with R is seen. This key starting with R is not visible all the time. It only comes into play when bitcoin starts re-indexing. The key is created before re-indexing and it gets deleted at the end of it.

Re-indexing cannot be stopped midway as Bitcoin constantly looks for the presence of this key. To stop Bitcoin from re-indexing, simply delete this key. Though seriously not advised. Reindexing is curse.

The bitcoin server bitcoind reads a file bitcoin.conf which is present in the folder /Users/vijaymukhi/Library/Application Support/Bitcoin.

```
bitcoin.conf
rpcuser=bitcoinrpc
rpcpassword=mukhi
txindex=1
```

One line, txindex=1 is added to the conf file. This enables transaction indexing, which by default it is off. The bitcoind program reads this configuration file and reconfigures itself accordingly. It now enables transaction indexing. The program may ask you to run bitcoind using the command line option -reindex option, please follow orders.

You will have to wait for a very, very, very long time for the bitcoin server to finish creating a transaction index. So, don't do it. But you may have no choice as later-on the code insists that you re-index each and every transaction. Please do not set the option txindex to 1 if not needed. For now, set it on. We always set this option on.

There are many number of keys beginning with t

```
ch2318.py
import leveldb
cntt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")
for k,v in db.RangeIter():
    if k[:1] == 't':
        cntt = cntt + 1
print "cntt=%d" % cntt
```

Output

cntt= 197572584

This program counts the key-value pairs where the keys start with a t. There are over 197 millions of them, once again our last count October 2017 was a humongous 261321081. These programs do take an entire night to finish. The more you delay indexing transactions, the longer it takes to access them in the future. The Bitcoin universe has generated over a million transactions to date and every key represents a transaction hash value.

Decoding the t Key

```
ch2319.py
import leveldb
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")
for k,v in db.RangeIter():
    if k[:1] == 't' :
        print k[1:].encode('hex')
        print reversehash(k[1:].encode('hex'))
    exit()
```

Output

```
74000000012f066b66737acfa060f8b56ce8f980bc2e7b697c9ede11dd1985fbac
000000012f066b66737acfa060f8b56ce8f980bc2e7b697c9ede11dd1985fbac
acfb8519dd11de9e7c697b2ebc80f9e86cb5f860a0cf7a73666b062f01000000
```

This example displays the key of the first key-value pair that starts with a 't'. The first byte is a 't' so the byte is skipped and then the hash value is displayed in reverse form.

We are not surprised that blockchain.info recognizes the reverse hash value starting with acf as a valid transactions hash. This transaction is found in block number, 307846. Your mileage will vary but any blockchain explorer will validate this hash.

Decoding the Value Stored along with a Key Named t

```
ch2320.py
import leveldb
from cfuncs import *
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)
def doutput(f , i):
    bitcoinvalue = r8bytes(f)
    outputsriptlen = rvarint(f)
    scriptpubkey = rbytes(f , outputsriptlen).encode('hex')
def dinput(f , i):
```

```
prevtranhsh = rhash(f).encode('hex')
outputindex = rint(f)
inputscriptlen = rvarint(f)
sigpubkey = rbytes(f, inputscriptlen).encode('hex')
seqno = rint(f)
return (reversehash(prevtranhsh), outputindex)

db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")
thash = "f4184fc596403b9d638783cf57adfe4c75c605f6356fbc91338530e9831e9e16"
hash1 = '74' + reversehash(thash)
v = db.Get(hash1.decode('hex'))
print v.encode('hex')
(fileno, offset) = base128(v, 0)
(offsetb, offset) = base128(v, offset)
(offsett, offset) = base128(v, offset)
print "File Number %d" % fileno
print "offset block %d" % offsetb
print "offset transaction %d" % offsett
f = open("/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat" % fileno, "rb")
f.seek(offsetb + 80 + offsett, 0)
start = f.tell()
print "Start %d" % start # 38255
tver = rint(f)
noinputs = rvarint(f)
for i in range(0, noinputs):
    (hash, index) = dinput(f, i)
nooutputs = rvarint(f)
for i in range(0, nooutputs):
    doutput(f, i)
locktime = rint(f)
end = f.tell()
f.seek(-(end - start), 1)
tran = f.read(end - start)
hash = chash(tran)
print reversehash(hash.encode('hex')) == thash
```

Output

```
0081a8188007
File Number 0
offset block 38040
offset transaction 135
Start 38255
True
```

The details of block number 170 are displayed in the website blockchain.info and it shows that the first block has two transactions. There is one Coinbase transaction and the second transaction has a hash value of f4184.

The variable thash is a hardcoded value of the transaction hash, as seen in the blockchain explorer. Unfortunately for us,

this hash value must be reversed. Thereafter, 0x74 or a 't' is added to the start of the hash. This hash is decoded and now it acts like our key.

There are three entities that make up the key value, all of them base128 encoded. The first number is the file number the transaction resides in. In the case of block 170, obviously, it will be found in blk00000.dat. The second number is the offset from the beginning of the file. A small correction. This offset is not from where the block starts, but 8 bytes from the start, 38040. At this point, the 80-byte block header is present.

Start any hex editor and look at the file blk00000.dat. We key in the number 38032 i.e. 38040 - 8 and there it is, the magic number of the block.

The offset, however, for some reason does not believe that the magic number and size are part of the block. The offset represents the start of the block header. In our code, we open the blk file based on the file number stored in variable fileno. The offset in the file where the block header starts is stored in variable offsetb. The last value in the key-value record is the offset from the start of the block header (this is important) where the concerned transaction starts, this is the offsett variable.

Three different values are used to move to the start of the transaction beginning with f418 in the file blk00000.dat. First, there is a jump to the start of the relevant block in the file, this value is stored in variable offsetb. One file, many blocks. Then the constant 80-byte block header is skipped.

Finally, let's jump to a transaction in that block. The offsett variable which contains the start of the transaction relative to the start of the block and not relative to the file. It positions the pointer at the actual start of the transaction where the variable length byte reveals the version number of this transaction.

As before, this starting point is saved in a variable called start. Then using the same code of the transaction chapter, one transactions inputs and outputs are read in two for loops. This is possible because the transactions are stored sequentially, starting with the number of transactions. It must be noted that we are not interested in reading all the transactions and we do nothing with the inputs and outputs we have read.

Once all the inputs and outputs and the locktime are read, the end-point is stored in the variable end. The entire transaction is read as we have no idea how large a transaction is. The length of a transaction is determined by using the end and start variables.

The 't' key gives access to a transaction with a minimal amount of fuss. We jump directly to the first byte of the transaction. Imagine reading even a single .dat file. Too slow. There are over 300 million transactions, at this point in time. At the end, we compare the computed transaction hash value with that of the key, after omitting the 't'. They match.

The transaction indexing adds a transaction to a block, it remembers the block file and the position in the block where it has stored the transaction.

Two fields would have sufficed, the start from the beginning of the file where the transaction starts. It uses one more field where it stores the start point of the block from the beginning of the file; and then where the transaction starts relative to the block header and not start of block.

Checking the 300 Million Keys Stored in the Index Folder

```
ch2321.py
import leveldb
from cfuncs import *
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
```

```
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def doutput(f , i):
    bitcoinvalue = r8bytes(f)
    outputscripflen = rvarint(f)
    scriptpubkey = rbytes(f , outputscripflen).encode('hex')

def dinput(f , i):
    prevtrhash = rhash(f).encode('hex')
    outputindex = rint(f)
    inputscripflen = rvarint(f)
    sigpubkey = rbytes(f , inputscripflen).encode('hex')
    seqno = rint(f)
    return (reversehash(prevtrhash) , outputindex)

db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index")
cnt = 0
for k,v in db.Rangelter():
    if k[:1] == 't' :
        (fileno , offset) = base128(v , 0)
        (offsetb , offset) = base128(v , offset)
        (offsett , offset) = base128(v , offset)
        f = open("/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat"% fileno, "rb")
        f.seek(offsetb + 80 + offsett , 0)
        start = f.tell()
        tver = rint(f)
        noinputs = rvarint(f)
        for i in range ( 0 , noinputs):
            (hash , index) = dinput(f , i)
            nooutputs = rvarint(f)
            for i in range ( 0 , nooutputs):
                doutput(f , i)
            locktime = rint(f)
            end = f.tell()
            f.seek(- (end - start) , 1)
            tran = f.read(end - start)
            hash = chash(tran)
            thash = reversehash(k[1:].encode('hex'))
            if reversehash(hash.encode('hex')) != thash:
```

```

print "Error in key %s:%d" % (k.encode('hex'), cnt)
exit()
cnt = cnt + 1
if cnt % 1000000 == 0:
    print cnt
    f.close()
print "Total Number of transaction keys pairs are %d" % cnt

```

Output

Total Number of transaction keys pairs are 196589280

Now let's decipher just one key-value pair starting with 't'. What works for one transaction key will work for all transaction keys in a loop.

In the for loop, the if statement filters out only those keys that start with a t. After reading the file number, offset of the block from the start of the file and the offset of the transaction from that block, we jump to the start of the transaction data using offsetb, offsett and the block header size 80.

The block hash value is computed and compared with the transaction hash key, removing the 't' at the beginning. The code is simply copied from the earlier program.

What is significant here is that the index folder is created with the data found in the .dat files in the blocks folder. Therefore, one cannot simply copy the index folder from one machine to another, the entire Bitcoin folder must be copied.

Try this out, run this program on a blk files copied from another machine, but with a different index folder and see the errors. Tried and tested.

The index folder allows quick access to any block or transaction stored in a dat file. It stores the offsets of where these entities start in what file. Access is made faster; the volumes are however very large.

To date, we have never ever executed a program that goes through over a 300 million key-value pairs. The above program scans through every key that starts with a 't' and then jumps to the file on disk where this transaction resides. It then finds the double Sha-256 hash value and compares it with the transaction hash value in the key. If there is an error, the program quits.

This program could have taken weeks to complete, but it only took days. A tribute to the speed of leveldb. What takes time, is the calculating the hash value of each transaction and jumping around the various dat files on disk. Though there may be mistakes, but in the end, the code works or we thought so.

```

Error in key
1095109f06419abb985fb8f15b105b1fb09b774d681cd1dfbf9ec68d1007000074:100

```

We reversed the hash value in the error message and we were wondering why we get an error after running under 200 million transactions. The fault dear Brutus is not in the stars but of segregated witnesses which causes multiple hard forks. The date of this transaction is new, its 3rd October 2017. Please confirm this date in a blockchain browser.

Segregated witnesses re-write the blockchain rules of calculating a hash. We have a separate chapter at the end of the book where we explain how a segregated witness transaction is different. Whenever our code does not work, we blame witnesses. It's too early to rewrite this code to handle witnesses.

CHAPTER 24

UTXO Dataset

Statutory Warning. Cigarette smoking is bad for health and so is reading this chapter. The Bitcoin Core developers realized that the UTXO set was extremely difficult to understand so from Bitcoin Core version 0.15 onwards, they simplified the model used to store unspent outputs. Our honest advice, please read about the new UTXO format in a later chapter.

However, please read this chapter to understand how complex a model can be. This is the bane of writing books; entire chapters go into the trash can when versions change. The silver lining is that some Bitcoin forks like Bitcoin Cash or Segwit2x refuse to move to the newer UTXO model, which was made obsolete from Version 0.14 onwards. For this chapter, you must use a version of Bitcoin that is 0.14 or lower.

The UTXO or the Unspent Transaction Output is the Holy Grail of the Bitcoin world. In our opinion, blockchain is synonymous with UTXO. The documentation of the UTXO file format is available only in the source code. Google comes very short here.

Let's start with the basics of the UTXO and then understand why the UTXO is the biggest innovation ever of the Bitcoin world. In the main Bitcoin folder, there is a sub-folder called chainstate. The UTXO's are stored in this folder.

You must make sure that both Bitcoin-Qt and bitcoind are not running. Then make a copy of the chainstate folder, we copy it to our Desktop. The same protocol is adopted for the index data set also. Simple rule of life, never work on the original data as the software places invisible locks in the folders.

The chainstate folder is about 2.7++ GB. It simply suggests that there will be a very large number of name-value key pairs in this leveldb folder.

Before we really start, we create two folders on our Desktop, one is called chainstateold for a UTXO set created by an older version of the bitcoin client and the other is chainstatenew for the newer client version. We will be working on the newer chainstate folder but not the latest.

Tabulating all the Keys in the Chainstate Folder

```
ch2401.py
import leveldb
#db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstateold")
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
cnt = cntc = cnto = cntb = cnte = 0
for k,v in db.RangeIter():
    cnt = cnt + 1
    if k[:1] == 'c' :
        cntc = cntc + 1
    elif k[:1] == 'B':
```

```

cntb = cntb + 1
elif ord(k[:1]) == 14:
cnto = cnto + 1
else:
cnte = cnte + 1

print "Total Number of keys %d" % cnt
print "Total Number of c keys %d" % cntc
print "Total Number of B key %d" % cntb
print "Total Number of obfuscate_key %d" % cnto
print "Total Number of extra keys %d" % cnte

```

Output

```

Total Number of keys 17254454
Total Number of c keys 17254452
Total Number of B key 1
Total Number of obfuscate_key 1
Total Number of extra keys 0

```

The chainstate folder is a leveldb folder that contains millions of key-value pairs. This is in a sense, like the index folder. Fortunately for us, the UTXO set mostly contains one key that starts with a c. Most people unofficially refer to this key as a coin. Then there is only one key that starts with a B and once again only one key that starts with a 0xe or a 14, which we see as a key named obfuscate_key. There are over 51 million transactions or leveldb records in our chainstate database in October 2017.

In the program, we use multiple if statements to count these 3-different key-value pairs. The last else statement does not get called at all which reaffirms that there are only 3 different key value pair types in the UTXO dataset.

The output in the older chainstate folder, chainstateold is as below.

```

Total Number of keys 11749737
Total Number of c keys 11749736
Total Number of B key 1
Total Number of obfuscate_key 0
Total Number of extra keys 0

```

The key obfuscate_key is simply missing. Why?

Understanding a Key Called obfuscate_key

```

ch2402.py
import leveldb
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
cnt = 0
xorkey = ''
for k,v in db.RangeIter():
    if ord(k[:1]) == 14:
        print "key string %s" % k
        print "key encoded %s" % k.encode('hex')
        print "value encoded %s" % v.encode('hex')

```

```
print "cnt %d" % cnt
xorkey = v[1:]
break
cnt = cnt + 1
```

Output

```
key string obfuscate_key
key encoded 0e006f62667573636174655f6b6579
value encoded 084e139eddd26ba610
cnt 0
```

The very first key in the UTXO set is called obfuscate_key. The value of this key starts with a number 8 denoting the length of the key. The next 8 bytes or 16 digits is a xor key. The length of the key obfuscate_key is only 13 bytes but there is a 0 preceding these bytes. Therefore, the first byte of the key is 14, 13 + 1 or 0x0e, the total length of the key. Both key and value have the same structure, it starts with the length, followed by the bytes.

The variable cnt has a value of 0 because the first key-pair in the leveldb folder is obfuscate_key.

For security reasons, the value of the UTXO set is Xored by this key and then stored. If the values of these keys beginning with c are to be deciphered, they must be XORed again with the same key.

If and when the Bitcoin blocks are re-indexed, the value of the key obfuscate_key changes. This xor key must be the first key-value pair in the UTXO set. The XOR key is a must to decrypt the value of the key. The current value of this XOR key is 08c72ef467d9a67a60, it keeps changing.

De-ciphering the Last Block Hash Key or the B Key

```
ch2403.py
import leveldb
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def xorstring(v, xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
        if k == 8:
            k = 0
    return finalstr
#db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstateold")
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
xorkey = ''
for k,v in db.Rangelter():
    if ord(k[:1]) == 14:
        print "key string %s" % k
        print "key encoded %s" % k.encode('hex')
```

```

print "value encoded %s" % v.encode('hex')
xorkey = v[1:]
if k == "B":
    print "First Byte %s " % k
    print "Value encoded before xoring %s" % v[:-1].encode('hex')
    if xorkey == '':
        finalstr = v[:-1].encode('hex')
    else:
        finalstr = xorstring(v, xorkey)
        finalstr = reverseahash(finalstr)
    print "Value encoded after xoring %s" % finalstr
    break

```

Output

```

key string obfuscate_key
key encoded 0e006f62667573636174655f6b6579
value encoded 08c72ef467d9a67a60
First Byte B
Value encoded before xoring 607aa6d967f42ec7612959a28203f0434a22393b50abcace32d0b3bc7cd62913
Value encoded after xoring 00000000000000000153ff7be5f7de842a589fe2375fe40952aa15651b2207d4

```

One cannot confirm that the UTXO set being read, will have the key called `obfuscate_key`. It all depends on the Bitcoin Core version being used. We are handling both these cases, presence and absence of the key.

A variable called `xorkey` is created with its value set to a null string. Then in a for loop, all the `leveldb` key-value pairs are read. There is a check for a key beginning with `0x0e` or a value of 14 as the first byte of the key.

If a match is found, then it is an obfuscated chainstate. The last 8 bytes are stored into the variable `xorkey`. There is nothing to de-obfuscate as this key is the first key. This key will un-obfuscate the value.

By reading the source code, we learnt that the key beginning with a B denotes a block number. This key represents the last block considered while creating the UTXO set. The value of this key is a block hash value. We always thought that a valid block hash starts with a series of 0's. This also proves that the value is Xored.

Then, there is a function called `xorstring` which is given the value and the `xorkey`. The function returns the original value. The if statement checks if the value of the variable `xorkey` is null. If yes, then the chainstate is not obfuscated at all and the value simply must be encoded.

If the XOR string has a value, the `xorstring` function is called which returns the original value. This value is then reversed.

Let's now focus on XORing a string. The function `xorstring` simply iterates through every byte of the value passed in string `v`. Then a string, `finalstring` is created by adding each byte that is un-XORed, for want of a better word. The value in the `finalstring` variable is then returned.

When we XOR (symbol \wedge) say 5 and 7, we get an answer of 2. But if we XOR 2 and 5 we get 7 and if we XOR 2 with 7 we get 5. This is the heart and soul of Xoring. The principle is that if we XOR two numbers A and B and save the answer C, we can XOR the answer C with A to get B or with B to get A. In this manner, the original un-Xored value can be obtained provided we have the key that Xored the two numbers.

The key is 8 bytes and the value `v` is larger than 8 bytes. So, variable `k` is incremented each time by 1 and when it reaches 8, it is reinitialized to 0. This variable `k` is used to extract individually bytes, one at a time from the XOR key.

The variable `j` identifies the value of the key-value pair. Two values are being Xored, one from the value passed and the other from the XOR key. This gives the original byte back.

The bytes are also concatenated to the variable `finalstr`. The variables `j` and `k` index the value and XOR string independently as the XOR string cannot be larger than 8.

When the same code is processed on a unobfuscated chainstate, the `XORstring` function is not called as our values are not obfuscated.

First Byte B

Value encoded before xoring

000000000000000002e3fb8f0e7249f27311a5b8d202e89abc2848f8306a7e9e

Value encoded after xoring

000000000000000002e3fb8f0e7249f27311a5b8d202e89abc2848f8306a7e9e

The hash value of the last output was of block 452138, a block created in the year 2017. The second hash is block number 421583 of the year 2016. This also confirms that our work on this book spans over 3 years. We worked with different versions of Bitcoin to explain an obfuscated and un-obfuscated chainstate. The last time we ran this code, the block hash was displayed for block number 484360 but this was with Bitcoin Core 0.15.

The Heart of the UTXO Key is the c or Coin Key

ch2404.py

```
import leveldb
```

```
def xorstring(v , xorkey):
```

```
    j = 0
```

```
    k = 0
```

```
    finalstr = ''
```

```
    while j < len(v):
```

```
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
```

```
        j = j + 1
```

```
        k = k + 1
```

```
        if k == 8:
```

```
            k = 0
```

```
    return finalstr
```

```
def displaychainstate(finalstr , hash):
```

```
    print("(%d)%s\n%s" % (cnt, hash , finalstr)
```

```
xorkey = ''
```

```
cnt = 0
```

```
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
```

```
for k,v in db.RangeIter():
```

```
    cnt = cnt + 1
```

```
    if cnt >= 4:
```

```
        exit()
```

```
    if k[1] == 'c':
```

```
        if xorkey == '':
```

```
            finalstr = v.encode('hex')
```

```
        else:
```

```
            finalstr = xorstring(v , xorkey)
```



```

hash = k[1:]
displaychainstate(hash[::-1].encode('hex'), finalstr)
elif ord(k[:1]) == 14:
    xorkey = v[1:]

```

Output

```

(3)01028820010b2a00367244680f6da18acd861a08f0a89cb3b49ab50e
0118dd986e59473732239d39cb3b8890bf32677719dd8933b05f6614f4020000

```

This program is the spine when understanding the 11 million+ UTXO set. Other than two keys, all the other keys start with a c. This simplifies everything as there is just one key to deal with.

If the first key starts with 14, then the entire value is saved in a variable called xorkey.

But if the key starts with a c, there is a check on the variable xorkey for it to be an empty string. If yes, then our values are not Xored. If no, then the key's value is XORed before it can be used. This process with 11 million keys will take some time. In either case, the finalstring variable will contain the original unaltered value.

All processing happens in the function displaychainstate which is called for each key - pair beginning with a c. This function is passed two values. The first value is the actual key's value minus the first byte, c. The second parameter is the value which is the hash of a transaction, in this case a hash value starting with 011. The blockchain explorer reveals that this transaction has taken place in the year 2017. The cnt variable indicates that it is the third key in the UTXO set.

The value of the key in the UTXO set needs lots of explaining but the key itself is a transaction hash value. One hint, even though this transaction is of the year 2017, the second output is spent, but not the first. You will never get this hash value, but it will have unspent outputs or else it does not belong to the UTXO set.

The above line has its weight in gold. Only those transaction hashes will be found in the utxo set which have at least one unspent output.

Our order of transactions in the UTXO set will be different from yours. This UTXO set is dynamic, as all outputs in the transaction hash get spent, the hash is removed from the set. At the same time, as newer outputs get created which are unspent, these transactions get added to the UTXO set. Another name for the UTXO set is flux.

Ordering the Keys in a Leveldb Database

```

ch2405.py
import leveldb
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
for k,v in db.RangeIter():
    cnt = cnt + 1
    if cnt >= 8:
        exit()
    print k.encode('hex')

```

Output

```

0e006f62667573636174655f6b6579
42
6300000662484b459022b01156d71a584d67654e859c9da2566e155a887a46c9e1
630000007bd67bde0c5076561d27eb1a716abbe6c008a2f59b1240a2565adfee05
6300000872d371c478e4f9cd6464bb7135ce19bfe4f0e8d90be76be27a0b176182

```

```
6300000921dc8905f8e7fe039c86ac833c5bfdf749c37347bbd718bc2db431816e
630000096367d19a2cb4035950eeb48d20f1888ddf6e5c3b5cc9b6960f3b356bc2
```

One observation and clarification. The leveldb database sorts the keys internally. Therefore, the first key is always the XOR starting with 14 and then the block key beginning with a capital B. This is followed by the c keys ordered.

We make one small change to the above program.

```
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
for k,v in db.Rangelter(reverse=True):
    cnt = cnt + 1
```

Output

```
63ffffffb8edc5fcafc9539b5d089f328bc837ffbd762b1414884b8ed12ddc83cc2
63ffffffb5d6d13a0e82915c9f4000dbdcf6a21133d6a5a750147a7a3c8ebde2c3
63ffffff9adb5ce08cc28b0dd7a0c3d3318836ae6833401718809163636c1bcb1b2
63ffffff9956d10919e138b12342b0ae529349c12d6a53548a58b88300f830f358b
63ffffff5a6624afd7e8295b55aa74e950be6986a54323a84ad2cca02d4d5de4092
63ffffff50ad896f5ec809f7bbe7a32a8b7b35e672c810bd2591187f5ca7063fb46
63ffffff4ae6f789a94948985934c7acf42dacbc422076f72a2b8be345c3b105f73
```

Here, the Rangelter function is given a parameter called reverse which is assigned a value of true. The default value is False. The keys are in a reverse order, so they now do not start with 0s, but with Fs.

Extracting the Version Number and the nCode Field from the c Key

```
ch2406.py
import leveldb
def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
        if k == 8:
            k = 0
    return finalstr

def displaychainstate(finalstr , hash):
    finalstr = finalstr.decode('hex')
    print "(%d)%s\n%s" % (cnt, hash.encode('hex') , finalstr.encode('hex'))
    print "Version %d" % ord(finalstr[0])
    print "nCode %d:%s" % (ord(finalstr[1]) , bin(ord(finalstr[1])))

xorkey = ''
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
for k,v in db.Rangelter():
    cnt = cnt + 1
```

```

if cnt >= 4:
    exit()
if k[:1] == 'c':
    if xorkey == '':
        finalstr = v.encode('hex')
    else:
        finalstr = xorstring(v , xorkey)
    hash = k[1:]
    displaychainstate(finalstr , hash[::-1])
elif ord(k[:1]) == 14:
    xorkey = v[1:]

```

Output

```

(3)0118dd986e59473732239d39cb3b8890bf32677719dd8933b05f6614f4020000
01028820010b2a00367244680f6da18acd861a08f0a89cb3b49ab50e
Version 1
nCode 2:0b10

```

Let's look at the first transaction starting with 0118d in our leveldb UTXO database. As always, this transaction must be reversed before displaying it.

This transaction is then displayed in the website blockchain.info. It is vintage 2017, whiskey tastes better when aged. In October 2017, for some reason the lone unspent output is yet unspent. The blockchain explorer reveals that the transaction has one input and two outputs. The inputs are of no concern as they indirectly reference outputs. In a UTXO set, we are only interested in unspent outputs.

Again, only outputs of a transaction hash that are unspent will find a place in the UTXO set. The transaction needs at least one unspent output. However, more the merrier.

This transaction contains two outputs, the first output is unspent, and the second output as of now, is spent. If the first output had been spent, then this transaction would find its way out of the UTXO set.

The key of UTXO set is the transaction hash and its value should have the unspent outputs. This value field must be stored in a very optimized manner as the number of transactions are very large. Here, every bit counts.

A transaction will have multiple outputs and at times, certain outputs will be spent and some unspent. Sometime later in the future, the unspent outputs will be spent.

The number of transaction hashes in this set is dynamic. The number of transaction keys it stores reduces as these outputs get spent. So, there must be a UTXO value field to identify the unspent outputs. But...

In other words, at some point in time, all outputs of a transaction will be spent. When this happens, the transaction hash key will be removed from the UTXO set. The number of keys increases when newer transactions having unspent outputs are added. At the same time, older transactions having no unspent outputs will be removed.

There are over 11 million key-value pairs in the UTXO set. Thus, the representation of the outputs will be in a compressed form, to save memory and disk space. What is most fascinating is observing how the brightest minds in the world have created an optimized storage database for keeping track of transaction outputs that are not spent.

A transaction hash enters the UTXO set as it is newly mined. Its outputs get spent at some later point in time. A transaction is dropped from the UTXO set when there are no unspent outputs. There will never be a case where this transaction hash will ever re-enter the UTXO set. Once you are kicked out of the UTXO club, no reentry. Unlike a credit card transaction, a Bitcoin transaction can never be reversed.

We are not aware of any formal documentation or paper on the structure of the UTXO set. No source code, no chapter on UTXOs.

Coming back to code, the first value of the first transaction hash used a key. Like all entities in the Bitcoin world, the first byte is the version number 1. No wasting 4 bytes to store this version number, just 1 byte needed.

Again, the transaction hash key value starts from the second byte onwards of the key. This hash value once again is sent reversed as a parameter. The variable `finalstring` stores the actual unadulterated value of the key. The next byte of the value field is called `nCode` by the Bitcoin source code. There are a series of comments with two examples only that offer some clues on the structure of the UTXO value field.

This variable `nCode` has a value of 2 in our case, which signifies that the second bit of an entity is on. If the first bit is on, it is a Coinbase transaction. The second bit reveals whether the first output is spent/unspent, the third bit decides whether the second output is spent/unspent. This approach gets very convoluted. Then life gets more complicated further. The drawback is that transactions can have many outputs and using 1 bit for each output state, spent or unspent, is very inefficient, space wise.

The Bitcoin world uses a space saving method so our code must be intelligent enough to parse this value field.

Technically, the value in `nCode` field should give a count on the number of unspent outputs. When the `nCode` value is 10 in binary, the second bit is 1 so the first output is unspent.

From the value field, the amount or value of the Bitcoins and the Bitcoin address or the RipeMD hash value of every output are very important. Also, the height of the transaction or the block number of the transaction is crucial. If the hash value of the block is stored, it will take up too much space.

The value in the `nCode` field is displayed in binary form using the `bin` function. If the first bit is 0, it is not a Coinbase transaction and so on. The value displayed is 010.

Let's now decipher the value field of the first transaction hash by hand. The value in the value field is :

```
01028820010b2a00367244680f6da18acd861a08f0a89cb3b49ab50e
```

```
01 version
```

```
02 nCode
```

```
8820 Let's assume that these digits represent the amount of the unspent output.
```

```
01 This represents the type of output following.
```

```
0b2a00367244680f6da18acd861a08f0a89cb3b4 This is the 20 byte RipeMD hash value, the output is confirmed by blockchain.info.
```

```
9ab50e This represents the height of the block that the transaction falls under. Once again, it should be read as 449294.
```

The version, `nCode` and the RipeMD hash looks okay. The amount and the hash do not look right. The reason being all numbers that are present in the UTXO set are stored compressed, to save on space. The amount is compressed twice.

If the value in `nCode` is not shown as 2, do not despair. Keep increasing the `cnt` variable in the `if` statement comparison. Change this constant value to 5 to 6 etc. At some point in time, you will see the `nCode` value is the same as ours.

Decoding One Entire Value of the c Coin Key

```
ch2407.py
```

```
import leveldb
```

```
def base128a(ans , offset):
```

```
    n = 0
```

```

while True:
    chData = ord(ans[offset:offset + 1])
    offset = offset + 1
    print "chData %x offset %d %d:%d n %d %d:%x" % (chData , offset , n << 7 , n * 128 , n , chData & 0x7F ,
    chData & 0x7F)
    n = (n << 7) | (chData & 0x7F)
    print "n %d chData & 0x80 %s" % (n , chData & 0x80)
    if chData & 0x80 == 128:
        n = n + 1
        print "New value of n %d:%x" % (n , n)
    else:
        print "Value in last else n %d" % (n )
        amt = decompressamout(n)
        print "decompressed amt is %d" % amt
        return (amt, offset)

def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n

def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ""
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
        if k == 8:
            k = 0
    return finalstr

def base128(ans , offset):
    n = 0
    while True:

```

```
    chData = ord(ans[offset:offset + 1])
    offset = offset + 1
    n = (n << 7) | (chData & 0x7F)
    if chData & 0x80 == 128:
        n = n + 1
    else:
        return (n,offset)

def displaychainstate( finalstr , hash):
    finalstr = finalstr.decode('hex')
    print "(%d)%s\n%s" % (cnt, hash.encode('hex') , finalstr.encode('hex'))
    (version , offset) = base128(finalstr, 0)
    print "Version %d offset %d" % (version , offset)
    (nCode, offset) = base128(finalstr, offset)
    print "nCode %d offset %d" % (nCode , offset)
    (amt , offset) = base128a(finalstr, offset )
    print "amt %d offset %d" % (amt , offset)
    hasht = ord(finalstr[offset:offset+1])
    print "hasht %d" % (hasht)
    ripemdoffset = 21
    ripemd = finalstr[offset + 1 : offset + ripemdoffset]
    print "ripemd hash %s" % ripemd.encode('hex')
    (height , offset) = base128(finalstr, offset + ripemdoffset )
    print "height %d offset %d" % (height , offset)

xorkey = ''
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
for k,v in db.RangeIter():
    cnt = cnt + 1
    if cnt >= 4:
        exit()
    if k[:1] == 'c':
        if xorkey == '':
            finalstr = v.encode('hex')
        else:
            finalstr = xorstring(v , xorkey)
        hash = k[1:]
        displaychainstate(finalstr , hash[:-1])
        elif ord(k[:1]) == 14:
            xorkey = v[1:]
```

Output

```
(3)0118dd986e59473732239d39cb3b8890bf32677719dd8933b05f6614f4020000
01028820010b2a00367244680f6da18acd861a08f0a89cb3b49ab50e
Version 1 offset 1
nCode 2 offset 2
chData 88 offset 3 0:0 n 0 8:8
```

```

n 8 chData & 0x80 128
New value of n 9:9
chData 20 offset 4 1152:1152 n 9 32:20
n 1184 chData & 0x80 0
Value in last else n 1184
decompressed amt is 132000
amt 132000 offset 4
hasht 1
ripemd hash 0b2a00367244680f6da18acd861a08f0a89cb3b4
height 449294 offset 28

```

This example breaks up the value of one transaction hash key. Obviously, there are several assumptions made which will be knocked off along the way.

All the fields starting from the version and the nCode fields are XOR encoded. Their current values are taken as face value.

Let's look at the number of Bitcoins for the unspent output only. The blockchin.info shows the Bitcoin value as 0.00132 BTC for the transaction hash starting with 0118d. What should catch attention is the unspent Bitcoin address, it starts with 1LpC.

The amount shown in the value field is nowhere close to this number. What adds to our woes is that along the way, this value changes twice. Amounts stored in the UTXO set are first encoded using base128 encoding. Base58 and Base128 encodings, to the best of our knowledge, are only used in the Bitcoin world.

The length of the encodings keep varying, so it is impossible to figure out how many bytes will be used to encode a number. The uphill task here is to recover the decoded number and the number of bytes the encoding takes up.

The function base128 and base128a are the same but there is one small addition at the end. This additional code reviews the amount, which requires an additional compression. The function decompressamt is called before returning the value. The print statements are added to enhance our understanding. The functions base128 and base128a are given the same two parameters, ans and offset. First is the value field as a decoded string and next is the starting point from where this string is to be read.

When the base128 function is called the first time, the value of the offset parameter is 0. The decoding is from the start of the string. The function base128 will return the decoded value and the number of bytes read for it. On paper at least, this is not an absolute number of bytes read but the starting point of the value data for the next field.

As the version and nCode field are not very simple, the while True loop is executed only once and the offset variable is incremented only once. At the end of the first function call, the offset variable (version field) has a value of 1. After determining the value of the variable nCode, the offset variable takes a value of 2. The variable n is initialized to 0 before the infinite while loop. This variable will finally contain the base128 decoded integer value. As it is vague as to how many bytes will be taken up by the encoding, the infinite while loop comes into play. A bane of variable length encodings.

At a time, the chData variable has one byte of the ans string. Its actual value will be returned using the ord function. The variable offset is the position from where the byte in the string is read. The offset variable is increased by one each time in the loop.

The variable is named as chData just to be in sync with the Bitcoin code. We adopt the approach of copying the algorithm and the variable names, just to stay true to the original code. Plus, on a more practical note, there are a few errors but nothing gets lost in translation.

The amount is decoded using the base128a function. The first print statement displays byte 88 and then 20. This is only because one byte is read at a time from the string ans starting from offset 2. The offset variable, true to form increases by 1.

A bitwise and with a 1 results in the same answer. So, when there is a bitwise and with 7f, all bits remain the same but the last or the highest bit, which becomes 0.

Another Summary.

Base128 encoding returns a variable length integer. There is no starting byte to state the number of bytes taken up by this integer. However, the high bit does give information on whether in the integer, it is the last byte or there are many more following.

The base128 encoding gives a number between 0-127 and takes up only one byte. Numbers between 128-16511 takes up 2 bytes and numbers from 16512-2113663 takes up 3 bytes.

Base128 encoding does not follow the C data types like char for 1 byte, short for 2 bytes and int/long for 4 bytes. These values are cast in stone. There is no upper range in a base128 encoding. Remember a Satoshi is a very small number, so, a language like C would have a problem representing it.

The only way to decipher the procedure used for decompressing is by first understanding the compression algorithm in the first place.

Following our logic, we start out with an Xored amount of 8820 and after decompressing, the value obtained is 132000. The blockchain explorer also agrees with the decompressing method. This mystery will take some time to resolve.

Let's now look at the variable hasht. The structure of a script hash in an output changes a lot. The source code reveals three cases. The ord function extracts this single byte. The current value of the variable hasht is 1. Next, the RipeMD hash value is read. We start reading 1 byte from the offset variable as the offset variable points to the script type.

20 bytes are read from the offset, though it looks like we are reading 21 bytes, but in the slice operator, the last byte does not count. The variable ripemdoffset ensures that our code runs in a loop later.

This is the simplest hash type we can ever meet.

Finally, the height of the block or the block number is read. To arrive at the starting point of this value, we add the offset variable (which still points to the hash type byte) and the ripemdoffset value (which is the length of the hash + 1 i.e. 21). The height of the block is 449294, the blockchain explorer confirms the same.

All this is great but how do we validate our workings. It is time consuming to check every hash key with the blockchain explorer blockchain.info. The whole process must be automated and plus we need someone else to concur our doings.

Decoding a Hash Type of pubkeyhash

```
ch2408.py
import leveldb
import subprocess
import json
def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
```



```

    n = n + 1
    else:
        amt = decompressamout(n)
        return (amt, offset)
def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n
def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
        if k == 8:
            k = 0
    return finalstr
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)
def displaychainstate(finalstr , hash):
    finalstr = finalstr.decode('hex')
    hash = hash[:-1]
    print

```

```
print("(%d)%s\n%s" % (cnt, hash.encode('hex'), finalstr.encode('hex')))
(version , offset) = base128(finalstr, 0)
print "Version %d offset %d" % (version , offset)
(nCode, offset) = base128(finalstr, offset)
print "nCode %d offset %d" % (nCode , offset)
(amt , offset) = base128a(finalstr, offset )
print "amt %d offset %d" % (amt , offset)
hasht = ord(finalstr[offset:offset+1])
print "hasht %d" % (hasht)
raw = subprocess.check_output(["bitcoin-cli", "gettxout", hash.encode('hex'), '1'])
if len(raw) == 0:
    return
print "raw %s" % raw
a = json.loads(raw)
typescr = a['scriptPubKey']['type']
if hasht == 0 and typescr == 'pubkeyhash':
    ripemdoffset = 21
    ripemd = finalstr[offset + 1 : offset + ripemdoffset]
    print "ripemd hash %s" % ripemd.encode('hex')
    (height , offset) = base128(finalstr, offset + ripemdoffset)
    print "height %d offset %d" % (height , offset)
    print "script/hash type %s" % typescr
    print "Whole scriptpubkey %s" % a['scriptPubKey']['hex']
    print "Usable scriptpubkey %s" % a['scriptPubKey']['hex'][6:-4]
    addrb = a['scriptPubKey']['hex'][6:-4] == ripemd.encode('hex')
    if addrb:
        print "Correct Value %s" % addrb
    else:
        print "Error in hasht == 0"
    exit(0)
xorkey = ""
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
for k,v in db.RangeIter():
    cnt = cnt + 1
    if cnt >= 5:
        exit()
    if k[:1] == 'c':
        if xorkey == "":
            finalstr = v.encode('hex')
        else:
            finalstr = xorstring(v , xorkey)
        hash = k[1:]
        displaychainstate(finalstr , hash)
    elif ord(k[:1]) == 14:
        xorkey = v[1:]
```

Output

(3)0118dd986e59473732239d39cb3b8890bf32677719dd8933b05f6614f4020000

01028820010b2a00367244680f6da18acd861a08f0a89cb3b49ab50e

Version 1 offset 1

nCode 2 offset 2

amt 132000 offset 4

hasht 1

(4)e1c9467a885a156e56a29d9c854e65674d581ad75611b02290454b4862060000

0104a7cf820700d957c2536c205e2483b635ce17b2e02036788d548ac970

Version 1 offset 1

nCode 4 offset 2

amt 9466355 offset 6

hasht 0

raw {

"bestblock": "00000000000000000000aac700522b874879f0416bd1f0aaeca63643bac79714c9",

"confirmations": 263788,

"value": 0.09466355,

"scriptPubKey": {

 "asm": "OP_DUP OP_HASH160 d957c2536c205e2483b635ce17b2e02036788d54 OP_EQUALVERIFY
OP_CHECKSIG",

"hex": "76a914d957c2536c205e2483b635ce17b2e02036788d548ac",

"reqSigs": 1,

"type": "pubkeyhash",

"addresses": [

"1LpCmEejWLNfZigApMPwUY9nZTS8NTJCNS"

]

},

"version": 1,

"coinbase": false

}

ripemd hash d957c2536c205e2483b635ce17b2e02036788d54

height 189808 offset 30

script/hash type pubkeyhash

Whole scriptpubkey 76a914d957c2536c205e2483b635ce17b2e02036788d548ac

Usable scriptpubkey d957c2536c205e2483b635ce17b2e02036788d54

Correct Value True

Fair weather warning. If the output is not what you see, then please replace the if statement condition `cnt >= 5` to `cut >= 6` and so on and of forth.

In the program, the value field, finalstring is first decoded and then the hash value is reversed. This process is now performed in the function and not outside it.

After displaying the value of the four field's data, version, nCode, amt and hash, its time to verify our acts in the eyes of the Bitcoin law. We use the same base128 function that we wrote earlier. This is optional.

The `getxout` command displays the output details of a transaction given the index or offset of the output within the transaction. It is the last parameter. In other words, this command simply reads the UTXO data set. The transaction

hash value and the output index are in a string format. In this case, the second transaction hash value starts with e1c9. The output index is hard coded to a value of 1. Our second output is unspent; you may have to write 0 or 2 if things do not pan out. You must write your unspent output index or else the rest of the output will not show. This will be automated later. The gettxout does not return an error, it simply does nothing, says nothing.

The string returned is converted into a json object using the loads functions as before. Strings do not allow easy access to fields, hence it is converted to a dictionary. The field called scriptPubKey is important here as it has a field called type, which gives the type of output hash. In this case, the type is pubkeyhash as the value of the hasht variable is 0. The raw variable is also displayed so that you can visually confirm the data.

As there will be different hasht types, 6 as per documentation, 5 in real life, an if statement comes handy to make sure that the hasht variable and script type match the gettxout command. One more error check. The if statement matches in our case.

The scriptpubkey field is displayed and within it, is the hex field for the entire script public key value. The hex field contains not only the standard opcodes but also the length of the RipeMD160 hash and the actual hash value.

For a pubkeyhash type, the first opcode will always be OP_DUP followed by OP_HASH160, then the length of the hash, the actual hash and final OP_VERIFY and OP_CHECKSIG.

Unfortunately for us, in the UTXO set, only the RipeMD hash is stored. This hash in the ripemd variable is extracted by reading 20 bytes from the finalstring string using the offset variable. In this manner, the RipeMD hash value is read from the output returned by the gettxout command.

The first standard 3 bytes or 6 digits (76 a9 14) and the last 2 opcodes (88 ac) or 5 bytes are skipped to get the ripemd160 hash. On an encoded string, these values are doubled. As before, the height stored at the end of the value field is also extracted.

A few values are displayed for clarity. What is significant here is the hex field where the entire script field and only the hash values are present. The next line displays only the extracted usable hash value.

Finally, the ripemd160 hash in the UTXO set is compared with the return value of the gettxout command. The addrb variable is printed which must be true if the task is performed correctly. If the else statement ever gets called, then something is wrong. At some point in time, we will run our code through 11 million keys to check all eventualities.

At times, the length of the variable raw is 0 and it is not an error. Though it's confusing, as we are reading the UTXO set which contains only unspent outputs. This happens only because when the chain state folder was saved on our Desktop, this output was still unspent.

As some days elapsed and some code was executed, and the UTXO set being dynamic, lots of unspent outputs become spent and hence get removed from the new UTXO set but remained in the old static on disk UTXO set. The UTXO set is outdated, its stale. We have to run the bitcoin server bitcoind in the background or else we cannot run the bitcoin client.

Therefore, we make sure that we check if the output of the gettxout command is a null string. It should not normally be the case. A simple check, if the length of the string returned in raw has a length of 0, then the output is spent and quit out. But no errors are to be reported.

Once again, the ambiguous assumption here is that, we have just one output in the value field.

The Hash Type being Checked for is Scripthash

```
ch2409.py
import leveldb
import subprocess
```

```

import json

def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)

def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1

```

```
k = k + 1
if k == 8:
    k = 0
return finalstr

def displaychainstate(finalstr , hash):
    finalstr = finalstr.decode('hex')
    (version , offset) = base128(finalstr, 0)
    (nCode, offset) = base128(finalstr, offset)
    hash = hash[:-1]
    outarr = []
    if nCode & 0x01 == 1:
        if 0 not in outarr:
            outarr.append(0)
    if nCode & 0x02 == 2:
        if 0 not in outarr:
            outarr.append(0)
    if nCode & 0x04 == 4:
        outarr.append(1)
    if nCode <= 7:
        for out in outarr:
            outs = "%s" % out
            raw = subprocess.check_output(["bitcoin-cli", "gettxout" , hash.encode('hex') , outs])
            if len(raw) == 0:
                return
            a = json.loads(raw)
            typescr = a['scriptPubKey']['type']
            (amt , offset) = base128a(finalstr, offset )
            hasht = ord(finalstr[offset:offset+1])
            if hasht == 1 and typescr == 'scripthash':
                print "typescr %s hasht %d" % (typescr , hasht)
                print "outarr %s out %d" % (outarr , out)
                print "hash %s" % hash.encode('hex')
                print "finalstr %s" % finalstr.encode('hex')
                print "a %s" % a
                print "amt %d" % amt
                ripemdoffset = 21
                ripemd = finalstr[offset + 1 : offset + ripemdoffset]
                print "a %s" % a['scriptPubKey']['hex']
                print "ripemd %s" % ripemd.encode('hex')
                print "a %s" % a['scriptPubKey']['hex'][4:-2]
                addrb = a['scriptPubKey']['hex'][4:-2] == ripemd.encode('hex')
                print "addrb %s" % addrb
            exit()
    xorkey = ""
    cnt = 0
    db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
```

```

for k,v in db.Rangelter():
    cnt = cnt + 1
    if k[:1] == 'c':
        if xorkey == '':
            finalstr = v.encode('hex')
        else:
            finalstr = xorstring(v , xorkey)
        hash = k[1:]
        displaychainstate(finalstr , hash)
    elif k[:1] == 'B':
        pass
    elif ord(k[:1]) == 14:
        xorkey = v[1:]
    else:
        print "Unknown found %s:%s" % (k.encode('hex') , v.encode('hex'))

print "Total Number is %d" % (cnt)

```

Output

```

typescr scripthash hasht 1
outarr [0] out 0
hash e8458a67bb32af5d170b5a1d2732c1510e13b34905e52afa58da5017880c0000
finalstr 010285ebb81501b7998352903fc3e8af09149b4d73fb624de29ab298bc2a
a {u'version': 1, u'value': 0.01595523, u'bestblock':
u'000000000000000001d163653aa9c183c244796cf24f443bcc886899dc785528',
u'confirmations': 30165, u'coinbase': False, u'scriptPubKey': {u'reqSigs': 1, u'hex':
u'a914b7998352903fc3e8af09149b4d73fb624de29ab287', u'addresses':
[u'3JR0UvAbrYVjyWmru36NmA5VuFNu7CBFqy'], u'asm': u'OP_HASH160
b7998352903fc3e8af09149b4d73fb624de29ab2 OP_EQUAL', u'type': u'scripthash'}}
amt 1595523
a a914b7998352903fc3e8af09149b4d73fb624de29ab287
ripemd b7998352903fc3e8af09149b4d73fb624de29ab2
a b7998352903fc3e8af09149b4d73fb624de29ab2
addrb True

```

The above program reads the UTXO database and it is the largest program we have written so far. However, it does not take the maximum amount of time to run. It has been broken up into smaller programs though with a few changes here and there.

The first three bits of the nCode variable are understood separately. The first bit if on, it signifies that the transaction as a coinbase transaction. A coinbase transaction is a way for the new Bitcoins to enter the system. All Bitcoins come from a coinbase output.

There is an empty array called outarr which contains the unspent outputs in this transaction. The assumption of only one unspent output in the transaction along with the output index number is not true anymore.

Let's see how this array, outarr is filled with unspent outputs.

When the first bit is on, a 0 is added to indicate that it is the offset of the output. This is important to the outarr array. The

append function of the array class is used for this purpose. However, first the array is checked to already contain the output index 0. No error, but why do something twice.

If the second bit is on, then again the index of the output is 0. Here the difference is that the transaction is not a coinbase transaction. In both cases, the first output is the unspent output.

When the third bit is on or the value is 4, the second output is the unspent output, so 1 is added to the outarr array.

Now the if clause comes in to attend to only the first three bits since there are many more still left to be taken care off. There will be a certain transaction which is not a coinbase transaction but has the first and second output is marked as unspent. Anything is possible.

An if statement is also required to take care of those key-value pairs where the nCode variable is less than 7. Plus, the case, where both outputs 0 and 1 are unspent. The program has only accounted for the first 3 bits; the other bits are more complex to handle.

In the for loop, the outarr array will contain a 0 or 1, or two members with the value of 0 and 1. The gettxout command needs a string so the array member is converted to a string using the %s print modifier. This is where the output index is dynamically taken care of.

The gettxout command returns scripthash, the type is script hash. The value of the hasht variable is 1 and not 0.

The length of the RipeMD hash is once again 20 bytes but there are a few changes. The hash type starts with the opcode a9 which is OP_HASH160, the same length byte 0x14 and it ends with an opcode 0x87 or OP_EQUAL.

Therefore, only the first 2 bytes and just the last byte, are extracted. There is one opcode less at the start and one opcode less at the end compared to the pubkeyhash type.

Nothing else changes.

Since we do not know as to which key in our UTXO output will have a transaction output type of scripthash, all the print statements are placed in the if statement. This if statement checks if the UTXO set gives a hash type of 1 and the gettxout command gives a value of scripthash. Placing the prints outside would bring in too much clutter on the screen. All this will go later.

Hash Type pubkey

```
ch2410.py
import leveldb
import subprocess
import json

def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)

def decompressamout(x):
```



```

    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
        if k == 8:
            k = 0
    return finalstr

def displaychainstate(finalstr , hash):
    hasht = ''
    typescr = ''
    finalstr = finalstr.decode('hex')
    (version , offset) = base128(finalstr, 0)
    (nCode, offset) = base128(finalstr, offset)
    hash = hash[:-1]
    outarr = []
    if nCode & 0x01 == 1:
        if 0 not in outarr:

```

```
    outarr.append(0)
    if nCode & 0x02 == 2:
        if 0 not in outarr:
            outarr.append(0)
        if nCode & 0x04 == 4:
            outarr.append(1)
        found = 0
        if nCode <= 7:
            ripemdoffset = 0
            for out in outarr:
                outs = "%s" % out
                raw = subprocess.check_output(["bitcoin-cli", "gettxout", hash.encode('hex'), outs])
                if len(raw) == 0:
                    return
                a = json.loads(raw)
                typescr = a['scriptPubKey']['type']
                (amt, offset) = base128a(finalstr, offset)
                hasht = ord(finalstr[offset:offset+1])
                if hasht == 0 and typescr == 'pubkeyhash':
                    ripemdoffset = 21
                    ripemd = finalstr[offset + 1 : offset + ripemdoffset]
                    addrb = a['scriptPubKey']['hex'][6:-4] == ripemd.encode('hex')
                    if (hasht == 2 or hasht == 3) and typescr == 'pubkey':
                        print "-----(%d) hash 2 or hash 3" % cnt
                        print "-----Hash %s" % hash.encode('hex')
                        print "finalstr %s" % finalstr.encode('hex')
                        print "cnt %d outarr %s out %d nCode %d" % (cnt, outarr, out, nCode)
                        print a
                        print "amt %d typescr %s hasht %d" % (amt, typescr, hasht)
                    ripemdoffset = 33
                    ripemd = finalstr[offset + 1 : offset + ripemdoffset]
                    addrb = a['scriptPubKey']['hex'][4:-2] == ripemd.encode('hex')
                    print "ripemd %s" % ripemd.encode('hex')
                    print "hex %s " % a['scriptPubKey']['hex']
                    print "hasht 2 or 3 addrb %s" % (addrb)
                    found = 1
                    offset = offset + ripemdoffset
                    (nHeight, offset) = base128(finalstr, offset)
                    if found == 1:
                        print "New offset is %d Start %s" % (offset, finalstr[offset:].encode('hex'))
                        print "Finally nHeight %d" % nHeight
                        exit()
xorkey = ""
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
for k,v in db.RangeIter():
```

```
print "Total Number is %d" % (cnt)
```

[illegible]

Let's take the 5001th key-value pair in our case only which has a transaction hash value starting with 39c3. The height of the block n the Bitcoin explorer matches the value of 438954.

When the hash type is 2 or 3, things are much simpler. The hex field starts with a 0x21 and it is followed by a 02. The last byte is 0xac which is the opcode for OP_CHECKSIG. These 3 bytes are removed to retrieve the RipeMD hash value. The blockchain explorer does not show the first byte 0x21, but starts with the 0x02 byte.

Handling a Hash Type of 4 or 5

```
ch2410a.py
import leveldb
import subprocess
import json

def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)

def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
```

```

        return (n,offset)
def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
        if k == 8:
            k = 0
    return finalstr
def displaychainstate(finalstr , hash):
    hasht = ''
    typescr = ''
    finalstr = finalstr.decode('hex')
    (version , offset) = base128(finalstr, 0)
    (nCode, offset) = base128(finalstr, offset)
    hash = hash[:-1]
    outarr = []
    if nCode & 0x01 == 1:
        if 0 not in outarr:
            outarr.append(0)
    if nCode & 0x02 == 2:
        if 0 not in outarr:
            outarr.append(0)
    if nCode & 0x04 == 4:
        outarr.append(1)
    if nCode <= 7:
        ripemdooffset = 0
        found = 0
        for out in outarr:
            outs = "%s" % out
            raw = subprocess.check_output(["bitcoin-cli", "gettxout" , hash.encode('hex') , outs])
            if len(raw) == 0:
                return
            a = json.loads(raw)
            typescr = a['scriptPubKey']['type']
            (amt , offset) = base128a(finalstr, offset )
            hasht = ord(finalstr[offset:offset+1])
            if (hasht == 4 or hasht == 5) and typescr == 'pubkey':
                print "——Hash %s" % hash.encode('hex')
                print "finalstr %s" % finalstr.encode('hex')
                print "cnt %d outarr %s out %d nCode %d" % (cnt , outarr , out , nCode)
                print a
                print "amt %d typescr %s hasht %d" % (amt , typescr , hasht)

```

```
ripemdoffset = 33
ripemd = finalstr[offset + 1 : offset + ripemdoffset]
addrb = a['scriptPubKey']['hex'][4:68] == ripemd.encode('hex')
print "hex %s " % a['scriptPubKey']['hex']
print "ripemd %s" % ripemd.encode('hex')
print "part %s" % a['scriptPubKey']['hex'][4:68]
print "hasht 4 or 5 addrb %s" % (addrb)
found = 1
if hasht == 0 and typescr == 'pubkeyhash' :
    ripemdoffset = 21
    ripemd = finalstr[offset + 1 : offset + ripemdoffset]
    addrb = a['scriptPubKey']['hex'][6:-4] == ripemd.encode('hex')
    if found == 1:
        print "--Hash type 0"
        print "Amount %d" % amt
        print "offset %d" % offset
        print "ripemd %s" % ripemd.encode('hex')
        print "hex %s " % a['scriptPubKey']['hex']
        print "pubkeyhash addrb %s " % (addrb)
        offset = offset + ripemdoffset
        (nHeight, offset) = base128(finalstr , offset )
        if found == 1:
            print "New offset is %d Start %s" % (offset , finalstr[offset:].encode('hex'))
            print "Finally nHeight %d" % nHeight
            exit()

xorkey = ''
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
for k,v in db.RangeIter():
    cnt = cnt + 1
    if k[:1] == 'c':
        if xorkey == '':
            finalstr = v.encode('hex')
        else:
            finalstr = xorstring(v , xorkey)
        hash = k[1:]
        displaychainstate(finalstr , hash)
    elif k[:1] == 'B':
        pass
    elif ord(k[:1]) == 14:
        xorkey = v[1:]
    else:
        print "Unknown found %s:%s" % (k.encode('hex') , v.encode('hex'))

print "Total Number is %d" % (cnt)
```

Output

```

---Hash 070ff3d8fe884d9a6dacdbaf59310bcba82ba565a44cf8a39559679b95450000
finalstr
010687eaf81305a39b9e4fbd213ef24bb9be69de4a118dd0644082e47c01fd9159d38637
b83fbc1100bd50abea6ea40816d832b989ca90eb0e75fdb50a8bf629
cnt 74 outarr [0, 1] out 0 nCode 6
{u'version': 1, u'value': 0.02060646, u'bestblock':
u'000000000000000001b49c61f04d87bf134213e44cd9500b7aa7581653a7c096',
u'confirmations': 235752, u'coinbase': False, u'scriptPubKey': {u'reqSigs': 1, u'hex':
u'4104a39b9e4fbd213ef24bb9be69de4a118dd0644082e47c01fd9159d38637b83fbc11
15a5d6e970586a012d1cfe3e3a8b1a3d04e763bdc5a071c0e827c0bd834a5ac',
u'addresses': [u'1VayNert3x1KzbpzMGt2qdqrAThiRovi8'], u'asm':
u'04a39b9e4fbd213ef24bb9be69de4a118dd0644082e47c01fd9159d38637b83fbc11
5a5d6e970586a012d1cfe3e3a8b1a3d04e763bdc5a071c0e827c0bd834a5
OP_CHECKSIG', u'type': u'pubkey'}}
amt 2060646 typescr pubkey hasht 5
hex
4104a39b9e4fbd213ef24bb9be69de4a118dd0644082e47c01fd9159d38637b83fbc11
5a5d6e970586a012d1cfe3e3a8b1a3d04e763bdc5a071c0e827c0bd834a5ac
ripemd a39b9e4fbd213ef24bb9be69de4a118dd0644082e47c01fd9159d38637b83fbc
part a39b9e4fbd213ef24bb9be69de4a118dd0644082e47c01fd9159d38637b83fbc
hasht 4 or 5 addrb True
---Hash type 0
Amount 2000000
offset 40
ripemd bd50abea6ea40816d832b989ca90eb0e75fdb50a
hex 76a914bd50abea6ea40816d832b989ca90eb0e75fdb50a88ac
pubkeyhash addrb True
New offset is 64 Start
Finally, nHeight 211881

```

This program illustrates the step-by-step approach taken to make our code more complete and complex. We hope that you get the same transaction hashes. The reason being that this transaction has both its outputs still unspent. The array outarr therefore, has two members, a 0 and 1. The nCode variable has a value of 6 which confirms the outputs 0 and 1 are both unspent.

The value field stored in the finalstring variable is extra-large. Let's break up the two outputs by hand and understand it better.

```

010687eaf81305a39b9e4fbd213ef24bb9be69de4a118dd0644082e47c01fd9159d38637
b83fbc1100bd50abea6ea40816d832b989ca90eb0e75fdb50a8bf629

01 version
06 nCode, 4 + 2 as there are 2 unspent outputs.
87eaf813 the first outputs amount 0.02060646 BTC. There is no reason to continue looking after finding a byte
whose highest bit is not 1.
05 Hash type 05 called pubkey
a39b9e4fbd213ef24bb9be69de4a118dd0644082e47c01fd9159d38637b83fbc hash size 64

```

The start of the second hash

```
11 amount value 0.02 BTC
00 hash type
bd50abea6ea40816d832b989ca90eb0e75fdb50a 40 byte second ripemd hash
8bf629 block height 211811
```

Once again, it is for the first time, there are two unspent outputs in one transaction. Let's hope none of these outputs get spent when you are trying the code. October 2017, still these outputs are not spent, maybe you will not so lucky.

Now, let's decipher the hash types of 4 or 5 or pubkey. In our case this hash type comes first in the code. First, 33 bytes are read from the value field. For the gettxout command, the first 2 bytes are dropped and then there is a read up to byte 34. This is to confirm that the hash values are the same. The output of the command gettxout is truncated to match the data stored in the UTXO database.

After the first output, comes the amount field of the second output. This has the value of 11. The hash type of 00 follows and then the actual hash. It ends with the height.

The heart of this program is the way the offset variable is computed. The variable ripemdooffset always contains the total number of bytes read after the amount field. It is a count/number of bytes taken up by each hash type. The current value of the offset variable is added to the ripemdooffset variable so that the amount field of the next output can be read. This is required as there will be multiple outputs. All outputs are stored back to back. An output only contains an amount and a hash type and the script.

Taking Care of a Multi-Signature Hash

```
ch2411.py
import leveldb
import subprocess
import json

def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)

def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
```



```

    x /= 9
    n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
        if k == 8:
            k = 0
    return finalstr

def displaychainstate(finalstr , hash):
    hasht = ''
    typescr = ''
    finalstr = finalstr.decode('hex')
    (version , offset) = base128(finalstr, 0)
    (nCode, offset) = base128(finalstr, offset)
    hash = hash[:-1]
    outarr = []
    if nCode & 0x01 == 1:
        if 0 not in outarr:
            outarr.append(0)
    if nCode & 0x02 == 2:
        if 0 not in outarr:
            outarr.append(0)
    if nCode & 0x04 == 4:
        outarr.append(1)
    if nCode <= 7:

```

```
for out in outarr:
    outs = "%s" % out
    raw = subprocess.check_output(["bitcoin-cli", "gettxout", hash.encode('hex'), outs])
    if len(raw) == 0:
        return
    a = json.loads(raw)
    typescr = a['scriptPubKey']['type']
    (amt, offset) = base128a(finalstr, offset)
    hasht = ord(finalstr[offset:offset+1])
    if typescr == 'multisig' and hasht <= 255:
        print "--Hash %s" % hash.encode('hex')
        print "finalstr %s:%d" % (finalstr.encode('hex'), len(finalstr))
        print a
        print "cnt %d outarr %s out %d nCode %d offset %d" % (cnt, outarr, out, nCode, offset)
        print "amt %d typescr %s hasht %d" % (amt, typescr, hasht)
        ripemdoffset = hasht
        ripemd = finalstr[offset + 1 : offset + ripemdoffset]
        print "hasht %d" % hasht
        print "ripemd %s" % ripemd.encode('hex')
        print "hex %s " % a['scriptPubKey']['hex']
        offset = 1
        while (True):
            print "ord %x" % ord(ripemd[offset])
            if ord(ripemd[offset]) == 0x21 or ord(ripemd[offset]) == 0x41 or ord(ripemd[offset]) == 0x36:
                offset = offset + 0x1 + ord(ripemd[offset])
            else:
                break
        print "Offset after while %d" % offset
        offset = offset + 2
        print "finalstr %s" % ripemd[0:offset].encode('hex')
        print "hex %s" % a['scriptPubKey']['hex']
        addrb = a['scriptPubKey']['hex'] == ripemd[0:offset].encode('hex')
        print "multisig addrb %s" % (addrb)
        exit()
xorkey = ""
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
for k,v in db.RangeIter():
    cnt = cnt + 1
    if k[1] == 'c':
        if xorkey == '':
            finalstr = v.encode('hex')
        else:
            finalstr = xorstring(v, xorkey)
        hash = k[1:]
        displaychainstate(finalstr, hash)
```

```

elif k[:1] == 'B':
    pass
elif ord(k[:1]) == 14:
    xorkey = v[1:]
else:
    print "Unknown found %s:%s" % (k.encode('hex'), v.encode('hex'))

```

Output

```

---Hash 55a3f75f26e40035612b3bdcec54afbeb45596eea0f24ad7318e68b023740000
finalstr
010487624d51210309a1d83fa88602e6ae72cd2679cdb813df4ee889ce47cb4b8ecdd60
7e44526fe211c434e545250525459000000000000000d806c1d500000042610c8405000
0000052ae92e92f:79
{u'version': 1, u'value': 1.25e-05, u'bestblock':
u'000000000000000003117568f0a3749fb6e98bf67f10848964ed06412c1fc0ee',
u'confirmations': 122728, u'coinbase': False, u'scriptPubKey': {u'reqSigs': 1, u'hex':
u'51210309a1d83fa88602e6ae72cd2679cdb813df4ee889ce47cb4b8ecdd607e44526fe
211c434e545250525459000000000000000d806c1d500000042610c84050000000052ae',
u'addresses': [u'1FBdB4b4HABBNz2Te2pEJz3zG7hFFyG5NY'], u'asm': u'1
0309a1d83fa88602e6ae72cd2679cdb813df4ee889ce47cb4b8ecdd607e44526fe
1c434e545250525459000000000000000d806c1d500000042610c840500000000 2
OP_CHECKMULTISIG', u'type': u'multisig'}}
cnt 117 outarr [1] out 1 nCode 4 offset 4
amt 1250 typescr multisig hasht 77
hasht 77
ripemd
51210309a1d83fa88602e6ae72cd2679cdb813df4ee889ce47cb4b8ecdd607e44526fe21
1c434e545250525459000000000000000d806c1d500000042610c84050000000052ae9
2e92f
hex
51210309a1d83fa88602e6ae72cd2679cdb813df4ee889ce47cb4b8ecdd607e44526fe21
1c434e545250525459000000000000000d806c1d500000042610c84050000000052ae
ord 21
ord 21
ord 52
Offset after while 69
finalstr
51210309a1d83fa88602e6ae72cd2679cdb813df4ee889ce47cb4b8ecdd607e44526fe211
c434e545250525459000000000000000d806c1d500000042610c84050000000052ae
hex
51210309a1d83fa88602e6ae72cd2679cdb813df4ee889ce47cb4b8ecdd607e44526fe21
1c434e545250525459000000000000000d806c1d500000042610c84050000000052ae
multisig addrb True

```

The script type multisig is the most difficult of all script types. This is a multi-signature script which carries with it multiple signatures. The length of this hash type is normally not larger than 255 bytes as normally a maximum of 3 signatures are used. In the entire UTXO set, we have not come across a multi signature type with 4 signatures.

The hasht variable has a value of 77 or 0x4d. It is a very large value representing a hash type. Nevertheless, the variable hasht's value is ignored for now. For the record, the length of the total string, finalstring is only 79 bytes. Contrast this with the value of the hash variable only. Therefore, we will not use the variable hasht when we are dealing with a multi-sig type.

The 79-byte finalstring looks like.

```
010487624d51210309a1d83fa88602e6ae72cd2679cdb813df4ee889ce47cb4b8ecdd60
7e44526fe211c434e54525052545900000000000000d806c1d500000042610c8405000
0000052ae92e92f
```

A break-up is given below

```
01 version
04 nCode
8762 amount
4d hash type
51 OP_1
21 public key first byte length
0309a1d83fa88602e6ae72cd2679cdb813df4ee889ce47cb4b8ecdd607e44526fe
21 public key first byte length
1c434e54525052545900000000000000d806c1d500000042610c840500000000
52 OP_2
ae OP_CHECKMULTISIG
92e92f height
```

The hex field from the gettxout command starts with byte 0x51 which is OP_1 and towards the end, the second last byte is 0x52 or OP_2. The last byte is 0xae or OP_CHECKMULTISIG.

As a quick revision, the multi signature script starts with 0x51, then a series of keys that start with the length 0x21 followed by 66 digits or 33 bytes. This ends with a OP_52 or at times a 0x53 or OP_3. The total number of public keys can vary.

When there are multiple signatures, only one or two out of three signatures can be used to sign the transaction to make it valid. Once again, the number of public keys cannot be predicted by anyone.

Let's now turn to code. The first byte 0x51 is ignored and an indefinite loop is placed where the offset variable has a value of 1 and not 0 to account for the byte 0x51. This structure remains the same. The idea is once the loop ends, the offset variable will have a count on the bytes. It is assumed that there are many more outputs following. We check for the various sizes of public keys.

These days everyone uses a compressed public keys of size 0x21, but in the good old days a larger uncompressed public key of size 0x41 was used. As a matter of fact, public keys of size 0x36 have also been found. All this has come to light while scanning 11 million plus transactions.

Now to compute the final offset variable. We first add 1 followed by the size of the public key. Now, there is no need to write multiple if statements to handle every different size of the public key encountered.

Thus, there is a hop skip jump to that many bytes into the multi signature string. If there is a byte other than these three lengths, the loop is exited. Normally this would be on a 0x52 or 0x53 which are denoted as OP_2 or OP_3 in the asm field.

The final offset variable will have to be 2 larger. This is to consider the last opcode OP_? and the multi signature opcode 0xae.

In this way, we keep a count of the bytes that make up the actual signature. This value we get is different from the value in variable `hasht`, 69 versus 77. As abundant caution, always check this value with the output received from the `gettxout` command. They must match with the chainstate data.

Types that Non-Standard

```

ch2412.py
import leveldb
import subprocess
import json
def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)

def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:

```

```
        return (n,offset)
def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
    if k == 8:
        k = 0
    return finalstr
def displaychainstate(finalstr , hash):
    hasht = ''
    typescr = ''
    finalstr = finalstr.decode('hex')
    (version , offset) = base128(finalstr, 0)
    (nCode, offset) = base128(finalstr, offset)
    hash = hash[:-1]
    outarr = []
    if nCode & 0x01 == 1:
        if 0 not in outarr:
            outarr.append(0)
    if nCode & 0x02 == 2:
        if 0 not in outarr:
            outarr.append(0)
    if nCode & 0x04 == 4:
        outarr.append(1)
    if nCode <= 7:
        for out in outarr:
            outs = "%s" % out
            (amt , offset) = base128a(finalstr, offset )
            hasht = ord(finalstr[offset:offset+1])
            raw = subprocess.check_output(["bitcoin-cli", "gettxout" , hash.encode('hex') , outs])
            if len(raw) == 0:
                return
            a = json.loads(raw)
            typescr = a['scriptPubKey']['type']
            if (typescr == 'pubkey' and hasht >= 6) or typescr == 'nonstandard':
                print "——Hash %s" % hash.encode('hex')
                print "finalstr %s" % finalstr.encode('hex')
                print "cnt %d outarr %s out %d nCode %d" % (cnt , outarr , out , nCode)
                print "nCode %d cnt %d " % (nCode , cnt )
                print "hasht %d" % hasht
                print a
            ripemdoffset = ord(finalstr[offset + 1]) + 3
```

```

    ripemd = finalstr[offset + 1 : offset + ripemdoffset]
    print "ripemd %s" % ripemd.encode('hex')
    print "hex %s" % a['scriptPubKey']['hex']
    addrb = a['scriptPubKey']['hex'] == ripemd.encode('hex')
    print "addrb %s" % addrb

xorkey = ''
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
for k,v in db.RangeIter():
    cnt = cnt + 1
    if cnt % 10000 == 0:
        print cnt
    if k[:1] == 'c':
        if xorkey == '':
            finalstr = v.encode('hex')
        else:
            finalstr = xorstring(v , xorkey)
        hash = k[1:]
        displaychainstate(finalstr , hash)
    elif k[:1] == 'B':
        pass
    elif ord(k[:1]) == 14:
        xorkey = v[1:]
    else:
        print "Unknown found %s:%s" % (k.encode('hex') , v.encode('hex'))

print "Total Number is %d" % (cnt)

```

Output

```

---Hash 3be0ac3dc1c3b7fa7fbc34f4678037ed733a14e801abe6d3da42bc643a651401
finalstr 010492d3080b76a90088ac889a27
cnt 69596 outarr [1] out 1 nCode 4
nCode 4 cnt 69596
hasht 11
{u'version': 1, u'value': 35.784, u'bestblock':
u'00000000000000000001a117780a9fa5901bf8ea25a8c420f6be128f5c90a257af',
u'confirmations': 296788, u'coinbase': False, u'scriptPubKey': {u'type':
u'nonstandard', u'hex': u'76a90088ac', u'asm': u'OP_DUP OP_HASH160 0
OP_EQUALVERIFY OP_CHECKSIG'}}
ripemd 76a90088ac889a27
hex 76a90088ac
addrb False

---Hash 22f995434ea5030a28a3be78c9dda72d5b63463734ce06cf218624aec6e62603
finalstr
0102052a22ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffac8df13c
cnt 201667 outarr [0] out 0 nCode 2

```

```
nCode 2 cnt 201667
hasht 42
{u'version': 1, u'value': 0.0001, u'bestblock':
u'000000000000000001e11ad1e722b8d9b650e4ab845db871baf395a88a7bbc0e',
u'confirmations': 203715, u'coinbase': False, u'scriptPubKey':
{u'type': u'pubkey', u'hex': u'22fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffac', u'asm':
u'ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff OP_CHECKSIG'}}
ripemd 22fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffac
hex 22fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffac
addrb True
```

Now let's take care of the tons of nonstandard transaction outputs that seem to have escaped the miner's attention. The first non-standard transaction hash value starts with 3be0. Patience is the key here as it takes many iterations to find it.

The gettxout command reports that type of this transaction is non-standard. Even blockchain.info cannot decode these outputs.

Each time the variable typescr reports a value of non-standard, it's time to break out of the loop. Such outputs make no sense at all. The other outputs, such as the asm key also makes no sense. Ignore these transactions for good. This is not what we are doing in this program.

One of the transaction despite having an output of type nonstandard was mined in block 150951 very successfully. It's output which is the second output in the list of outputs will however remain unspent as it has no valid Bitcoin address. It will always remain in the UTXO set for life. They cannot be removed due to which the UTXO set gets bulkier and filled with junk. They are many nonstandard hash types in the UTXO set.

There is an alternate blockchain explorer called <https://blockexplorer.com> if the standard one is down. We typed in the following transaction hash. 22f995434ea5030a28a3be78c9dda72d5b63463734ce06cf218624aec6e62603. It is shown as a valid transaction. Click on the + sign near the transaction hash and it shows the hex field with a length of 0x22 or 68 bytes. There are 68 F's following. Then we have the opcode 0xac or OP_CHECKSIG. Though they are not part of the standards scripts, our code must account for such hash outputs.

Once again this is technically not a non-standard transaction, but there is a hash type of pubkey. You can test the hashes in a Bitcoin explorer. Also, the variable cnt will have different values in your case.

Cross Checking the Amount and Height Using Bitcoin-cli

```
ch2413.py
import leveldb
import subprocess
import json
def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
```



```

    else:
        amt = decompressamout(n)
        return (amt, offset)
def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n
def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
        if k == 8:
            k = 0
    return finalstr
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)
def displaychainstate(finalstr , hash):
    finalstr = finalstr.decode('hex')
    hash = hash[:-1]
    print hash.encode('hex')
    (version , offset) = base128(finalstr, 0)

```

```
(nCode, offset) = base128(finalstr, offset)
(amt , offset) = base128a(finalstr, offset )
hasht = ord(finalstr[offset:offset+1])
raw = subprocess.check_output(["bitcoin-cli", "gettxout", hash.encode('hex'), '0'])
if len(raw) == 0:
    return
print "raw %s" % raw
a = json.loads(raw)
typescr = a['scriptPubKey']['type']
if hasht == 0 and typescr == 'pubkeyhash':
    ripemdoffset = 21
    (height , offset) = base128(finalstr, offset + ripemdoffset)

tmp = a['value'] * 100000000
amountb = tmp - float(amt) < 0.001
print "amt %d" % (amt )
print "a['value'] %1.12f" % a['value']
print "tmp %d" % tmp
print "tmp - float(amt) %d" % (tmp - float(amt))
print "amountb %s" % amountb
nHeights = "%s" % height
raw1 = subprocess.check_output(["bitcoin-cli", "getblockhash", nHeights ])
print "raw1 %s" % raw1
raw2 = subprocess.check_output(["bitcoin-cli", "getblock", raw1])
c = json.loads(raw2)
heightb = height == c['height']
print "height %d" % (height)
print "c['height'] %d" % c['height']
print "heightb %s" % heightb
print "No of transactions %d" % len(c['tx'])
unicodelist = c['tx']
asciilist = map(str, unicodelist)
tranb = hash.encode('hex') in asciilist
if tranb == False:
    print "The Transaction is not found %s" % (hash.encode('hex'))
else:
    print "Found transaction hash in block %s" % hash.encode('hex')
    for t in asciilist:
        if t == hash.encode('hex'):
            print "Found a transaction hash %s" % t
            break
    print c
    exit()

xorkey = ""
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
for k,v in db.Rangelter():
```

```

cnt = cnt + 1
if cnt >= 4:
    exit()
if k[:1] == 'c':
    if xorkey == '':
        finalstr = v.encode('hex')
    else:
        finalstr = xorstring(v, xorkey)
    hash = k[1:]
    displaychainstate(finalstr, hash)
elif ord(k[:1]) == 14:
    xorkey = v[1:]

```

Output

```
0118dd986e59473732239d39cb3b8890bf32677719dd8933b05f6614f4020000
```

```

raw {
  "bestblock": "000000000000000011ba57a127b54eaf6be67f93fd6b52bad3f696a66f4baeb",
  "confirmations": 4320,
  "value": 0.00132000,
  "scriptPubKey": {
    "asm": "OP_HASH160 0b2a00367244680f6da18acd861a08f0a89cb3b4 OP_EQUAL",
    "hex": "a9140b2a00367244680f6da18acd861a08f0a89cb3b487",
    "reqSigs": 1,
    "type": "scripthash",
    "addresses": [
      "32i3fvUTZkq2zeHBuosYDkiSCyMDhP62eo"
    ]
  },
  "version": 1,
  "coinbase": false
}

```

```
amt 132000
```

```
a['value'] 0.001320000000
```

```
tmp 132000
```

```
tmp - float(amt) 0
```

```
amountb True
```

```
raw1 000000000000000000271ce7ba96eb59cc1c4d7167609902a380df8f97616072f
```

```
height 449294
```

```
c['height'] 449294
```

```
heightb True
```

```
No of transactions 2690
```

```
Found transaction hash in block 0118dd986e59473732239d39cb3b8890bf32677719dd8933b05f6614f4020000
```

```
Found a transaction hash 0118dd986e59473732239d39cb3b8890bf32677719dd8933b05f6614f4020000
```

Here the focus is only on the first key-value pair in the UTXO data set. A gentle reminder, if you do not see more than two

lines of output, please change the '0' in the command `gettxout` to 1 or 2. So far, we have not double checked the amount and the height of the block.

The only problem with the base128 encoding is that 1 byte here or there and the height changes, though no error is reported. The height will yet be a valid height but smaller or larger than the actual value. This is an error on our part as the value field bytes are misinterpreted.

First, let's confirm that the amount has been decoded properly. The field called `value` present in the dictionary variable `a` is extracted. This variable `a` was returned by the command `gettxout`. The variable `tmp` has the amount of the unspent output plus more. This amount is however stored as a floating-point number whose value is 0.001320000000.

Please check that this value matches the first output shown in the blockchain explorer. The amount stored in the UTXO set is a decimal number in Satoshi, 132000. A floating-point number is compared with a number having no decimal places (but we know that it is has decimal places). The problem with floating point numbers is the accuracy. No one knows the final answer of 1/3.

So, the `amt` variable read from the UTXO set is converted to a float, using the `float` function. Then the `gettxout` value is multiplied by 100 million. The variable `a['value']` is multiplied by 10 raised 8 zeroes and the value is stored in the variable `tmp`. Then the answer is checked to be a very small number, not a 0. The amount difference is 0 in this case, but not all the time. We cannot compare floating point numbers like we compare integers.

Next in line is to confirm the block number or height. For this task given a block height, the block hash is obtained. The `getblockhash` method is used and it is passed a block height as a string. It was the first bitcoin-cli command used in this book. The variable `nHeights` is the height which is the last field of the UTXO fields value. Its value is 449294. The blockchain explorer confirms this value.

This block hash and the `getblock` method locates the actual block with all the transaction hashes. As is, running one external command using Python is slow, now 3 commands are used. There is no command in bitcoin-cli that takes a block height and returns the block details and here we need the block hash. The variable `c` contains all the block details. The field `height` gives the block height. We confirm that the height in the UTXO set matches the height returned by `getblock`.

As an aside, the block details also exhibits a list of transaction hashes in the block. This list is stored in the field `tx`. The `unicodelist` variable has this list of transaction hashes. For some reason, there is a Unicode/ascii error while comparing strings in the list.

The Python `map` function used in the code simply runs a function, the function name is given as the first parameter, `str`. It then executes this function on every member of the second parameter. In the program, the function is given the variable `unicodelist` that has all the transaction hashes. The `str` function converts Unicode strings into ascii, the documentation says this conversion is done automatically, but did not happen in our case and hence the use of `map`.

The `in` keyword checks if our transaction hash or string stored in the variable `hash` is in the list of transactions. This variable `hash` is part of the key. In our case, the transaction is in block 449294, so a value of `true` is returned.

To cross check, we scan every member of the `asciilist` array and check if our transaction hash is a member of this array. The result is the same as using the `in` keyword, the keyword though is faster and more optimized. You can avoid the last for loop we used.

Handling More than 2 Unspent Outputs in a Transaction

```
ch2414.py
import leveldb

def bitson(num , times):
```

```

times = times * 8
times = times + 1
arr = []
if num & 1 == 1:
    arr.append(1 + times)
if num & 2 == 2:
    arr.append(2 + times)
if num & 4 == 4:
    arr.append(3 + times)
if num & 8 == 8:
    arr.append(4 + times)
if num & 16 == 16:
    arr.append(5 + times)
if num & 32 == 32:
    arr.append(6 + times)
if num & 64 == 64:
    arr.append(7 + times)
if num & 128 == 128:
    arr.append(8 + times)
if num == 0:
    arr.append(0 + 2 + times)
return arr

def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)

def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):

```

```
n *= 10
e = e - 1
return n

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
        if k == 8:
            k = 0
    return finalstr

def displaychainstate(finalstr , hash):
    finalstr = finalstr.decode('hex')
    (version , offset) = base128(finalstr, 0)
    (nCode, offset) = base128(finalstr, offset)
    hash = hash[:-1]
    outarr = []
    if nCode & 0x01 == 1:
        if 0 not in outarr:
            outarr.append(0)
    if nCode & 0x02 == 2:
        if 0 not in outarr:
            outarr.append(0)
    if nCode & 0x04 == 4:
        outarr.append(1)
    increase = 0
    if nCode >= 8:
        print "----nCode %d:%s:%d cnt=%d" % (nCode , bin(nCode) , nCode >> 3 , cnt)
        print "hash %s" % hash.encode('hex')
        print "bytes %s" % finalstr.encode('hex')
        if (nCode & 0x02 == 0) and (nCode & 0x04 == 0):
            increase = 1
    cntvout = (nCode >> 3) + increase
```

```

print "cntvout %d offset %d increase %d" % (cntvout, offset , increase)
for i in range ( 0 , cntvout) :
    hexvout = ord(finalstr[offset:offset + 1])
    print "hexvout %x:%s" % (hexvout, bin(hexvout))
    while hexvout == 0:
        offset = offset + 1
    hexvout = ord(finalstr[offset:offset + 1])
    if hexvout != 0:
        break
    print "Out of the loop hexvout %x" % hexvout
    arrbits = bitson(hexvout , i)
    offset = offset + 1
    print "arrbits %s" % arrbits
    for bits in arrbits:
        outarr.append(bits)
    print "outarr %s" % outarr
    exit()

xorkey = ""
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstatenew")
for k,v in db.RangeIter():
    cnt = cnt + 1
    if k[:1] == 'c':
        if xorkey == "":
            finalstr = v.encode('hex')
        else:
            finalstr = xorstring(v , xorkey)
        hash = k[1:]
        displaychainstate(finalstr , hash)
    elif k[:1] == 'B':
        pass
    elif ord(k[:1]) == 14:
        xorkey = v[1:]
    else:
        print "Unknown found %s:%s" % (k.encode('hex') , v.encode('hex'))

```

Output

```

----nCode 16:0b10110:2 cnt=29
hash e53d983cefd03d48cf0aff66d16b8bebdaedb8bec5bd995c729bbec600220000
bytes
011630028ba443007cb9d3e17d19ee6fb4fa70aa5226886b547c44130f00750297f1ac9d
c15abb84bffe2a68072fd59332f90f00612e83d71cda8acb17a3e2141f6ddf4a0a80100d0
f00f1b76c285e16aafbc626cc4ba825b3001aea5350808b7a00c014c67d39c86f346b8b4
980576ca3229d7b7d2098b634
cntvout 2 offset 2 increase 0 (nCode >> 3) 2
hexvout 30:0b110000

```

```
Out of the loop hexvout 30
arrbits [6, 7]
hexvout 2:0b10
Out of the loop hexvout 2
arrbits [11]
outarr [0, 1, 6, 7, 11]
```

It's not over until the fat lady sings. This program figure out how outputs excluding the first or second, are encoded. It basically handles multiple unspent outputs. Also, the UTXO set due to its huge size, must be compressed in a very optimized manner.

The hash starting with e53d is first displayed. The site blockchain.info reveals that there are a total of 12 outputs; some spent, some unspent. These are the 0th, 1st, 6th 7th, 8th and the last or the 11th. They sort of match with the contents in the outarr list, which is outdated.

By the time, you come to this program, some more transaction outputs will be spent. If the unspent outputs decrease to 5, count yourself lucky. The eighth output was spent when we revisited this transaction hash in October 2017 so we are lucky. That's why our outputs shows you only 5 outputs and not 6.

The program checks if the second and third bits are on. In our case the value of nCode is 0x16 so, both bits 1 and 2 are on. This explains the initial two members of the list outarr, 0 and 1.

We now need to understand how the rest of the bytes point to certain output indexes.

The variable increase has a unique role to play. Its initial value is 0. First there is a check if the second and third bits are both 0's. If they are, then the increase variable changes its value from 0 to 1. In our case both the bits are 1 so the if statement is false and the increase variable stays at a value of 0.

The first three bits of the nCode variable check for the presence of the first two outputs. The right shift operator removes these three lower bits. The new value of nCode after the right shift by 3 will be 2. The value of the variable cntvout is now 2 as the increase variable has a value of 0. This cntvout variable indicate the number of bytes to be read from the finalstring variable.

The UTXO set has no idea on the number of unspent outputs from a transaction. This number will also decrease over time for certain transactions. The simplest and the most inefficient method would be to let every bit stand for a certain output index. This does not sound right.

So, what is the maximum number of outputs we should account for?

The value stored in variable nCode is right shifted by 3. This reveals the number of bytes to read next, which encode the output indexes.

If both the bits 1 and 2 are 0, then there must be at least one unspent output in the transaction. Therefore, one is added when the first two bits are 0's. Common sense says that there must be at the very least, one unspent output otherwise this transaction will not belong in the UTXO set.

The for loop is executed depending upon the number of bytes to read. In our case, it is only twice. The next byte read from the value field is 0x30.

The comments explicitly state that all bytes with a 0 must be ignored. The while loop is entered only when we hit a 0. If the next read byte does not have a value of 0, the while loop will break. The while loop was executed when the hexvout variable had a value of 0, now it has a non-zero value. This code simply jumps over the next 0s if present. It is an important check, as there will be many 0s.

To display the bits in this hexvout variable, a function called bitson is created. This function takes a byte and simply

discloses the on/off bits. Because 8 bits make a byte, there are a series of 8 simple if statements. In the bit pattern of a 0x30 or in binary 00110000, the fifth, sixth bits are on. The first 4 bits and the last two bits are all zeroes. The counting starts from 1 so one is added to get the right output index.

The function `bitson` simply uses if statements to check if the bit is on or off. It is better than the `bin` function because in the next round, the value of variable `i` will be 1.

Each time the function `bitson` is called, a value of 8 is added to each output index. Therefore, the times variable or the variable `i` is multiplied by 8. In the next iteration of the loop, the variable times will be 16. One more time, it will be 24.

In the second iteration of the loop, the `hexvout` variable is 2 or 00000010. The output is 2 + 8 (the times variable) + 1 (off by 1) or 11.

Once again. The `outarr` does not save the bit position but bit position + times. To account for the off by one error, 1 is added to the variable times.

The `arrbits` array contain the index of the outputs. We add them to the `outarr` array during each iteration of the for loop.

This is how multiple output indexes are represented in an optimized manner.

Compressing and Decompressing an Amount

`ch2415.py`

```
def decompressamount(x):
    print "--Decompressing"
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    print "x=%d e=%d" % (x, e)
    x /= 10
    print "x=%d" % x
    if (e < 9):
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
        print "d=%d x=%d n=%d" % (d, x, n)
    else:
        n = x + 1
        print "n=%d" % n
        while (e):
            n *= 10
            e = e - 1
        return n

def compressamount(n):
    if n == 0:
        return 0
    e = 0
    print "Start n=%d e=%d Conditions %s %s" % (n, e, n % 10 == 0, e < 9)
    while (((n % 10) == 0) and e < 9):
```

```
print "Start of loop n=%d e=%d" % (n , e)
n /= 10
e = e + 1
print "End of loop n=%d e=%d" % (n , e)

if e < 9 :
d = (n % 10)
n /= 10
yy = 1 + (n * 9 + d - 1) * 10 + e
print "If Statement True d=%d n=%d yy=%d" % (d , n , yy)
return yy
else :
aa = 1 + (n - 1)*10 + 9
print "n-1*10=%d" % ((n - 1)*10)
return aa

ans = compressamount(62900)
print ans
ans1 = decompressamout(ans)
print ans1
print "----"
ans = compressamount(12300000000000)
print ans
ans1 = decompressamout(ans)
print ans1
```

Output

```
Start n=62900 e=0 Conditions True True
Start of loop n=62900 e=0
End of loop n=6290 e=1
Start of loop n=6290 e=1
End of loop n=629 e=2
If Statement True d=9 n=62 yy=5663
5663
--Decompressing
x=5662 e=2
x=566
d=9 x=62 n=629
62900
----

Start n=12300000000000 e=0 Conditions True True
Start of loop n=12300000000000 e=0
End of loop n=12300000000000 e=1
Start of loop n=12300000000000 e=1
End of loop n=12300000000000 e=2
Start of loop n=12300000000000 e=2
End of loop n=123000000000 e=3
Start of loop n=123000000000 e=3
```

```

End of loop n=1230000000 e=4
Start of loop n=1230000000 e=4
End of loop n=123000000 e=5
Start of loop n=123000000 e=5
End of loop n=12300000 e=6
Start of loop n=12300000 e=6
End of loop n=1230000 e=7
Start of loop n=1230000 e=7
End of loop n=123000 e=8
Start of loop n=123000 e=8
End of loop n=12300 e=9
n-1*10=122990
123000
—Decompressing
x=122999 e=9
x=12299
n=12300
1230000000000000

```

The program shows how to compress the amount and then decompress it back to its original value. The original source is written in C++ but we converted it to Python here.

First the amount is compressed and then decompressed. Not the other way around. The steps taken in the compression phase, will be undone in decompression. The order is reversed. The first step in compression will be the last step in decompression.

The first task in the compress method is to bring in an error check. A value of 0 cannot be compressed.

The parameter `n` stands for the amount to be compressed.

In the while loop, there are two conditions to satisfy. The first one is the amount to be compressed, i.e. variable `n` must be divisible by 10. The mod of a number with 10 will only return 0 if that number is divisible by 10.

The second condition is that the value of variable `e` must be less than 9, there will be only 9 iterations in the loop. If you recall the 10 million we multiply by and how to convert a Satoshi to a Bitcoin. In our case as the amount passed in variable `n` is divisible by 10 and the value of `e` is 0, the while loop is true and therefore entered. The key condition is and, not or.

In the loop, the variable `n` is divided by 10. This simply knocks off one 0 from the equation. Then the value of the variable `e` is increased by 1. The while condition once again is true, the last 0 is dropped and the value of variable `e` is 2. The variable `n` has a value of 629.

The first condition of the while loop is false and the loop ends. The value of variable `e` being 2, gives a count of iterations or the 0s we knocked off, in our case 2 0's.

Now the if statement is true when the value of `e` is less than 9. The variable `d` is the last digit of our amount with the 0's removed. There is a mod by 10 on the variable so the value obtained will be less than 10 or the last digit. In our case, the variable `d` stores the last digit 9.

One more division of the amount by 10, drops the last digit 9. The initial value of 629 becomes 62. In effect, the amount is broken up into different numbers 62 and 9. The last digit and the rest. Join them and retrieve the original numbers.

Now we do something that is beyond our understanding. The amount is multiplied by the number 9. The value of variable

n is 62. Then the last digit stored in the number d which is 9 is added and 1 is subtracted from it. Further the value is multiplied with 10. For effect, 1 and the value in variable e are added; it has the number of 0's we originally removed, 2. In the end, a smaller value of 5663 is derived from 62900.

Now for the reverse or the decompression, the same rules are followed. A value of 0 cannot be decompressed as well. All steps of compression are reversed. The parameter name is x and not n.

We ended by adding a 1, so 1 is reduced and the value of the compressed amount reduces from 5663 to 5662. Then there is mod by 10 to access to the last digit 2 of the value in variable x. So variable e has a value of 2.

Same logic is applied here; the variable x is divided by 10 to get all but the last digit in the same variable x or the compressed amount. The variable x has a value of 566.

The if statement is true because the value of e is 2 which is less than 9. Here, the values are changed of 3 variables, namely d, x and n.

Earlier the amount was multiplied by 9 at the end, here there is a mod by 9. The subtraction by 1 is compensated by adding a 1. The value of d now becomes 9. The variable x is divided by 9 to gives a value of 62. Joining variables x and d results in 629.

These variables are not concatenated, instead a while loop is placed which will be iterated depending the value in variable e, which is 2. We multiply n by 10 and one is subtracted from the value in variable e.

The variable e has a count of the number of 0s removed from the original amount. So, in the while loop, the variable n is multiplied with that many 0s. The value in e is reduced by 1 each time.

While computing the value of yy in the compression stage, the value in e was added and then 1 was added to the answer. So, to arrive at the value of e, the compressed amount is reduced by 1 and then divided by 10 at the beginning of the function. The trick is that not more than nine 0s are removed. The mod of 10 gives a count on the number of 0s in the first place.

Let's confirm by taking a number larger than nine 0s, i.e. eleven 0s. Please count the number of loops, it must not exceed 9. The else is simpler. The value in n is reduced by 1 and multiplied by 10. The value of $(n-1) * 10$ is different as there are two 0s left still. There is an addition of 9 at the end because of nine 0s. Finally, a value of 1 is added to the returned answer.

At the time of decompression, the variable e has a value of 9 as 9 0s were removed in the compression process. A total of nine 0s are added at the end with the while loop. The else clause of the if statement gets called, and a value of 1 is added to the variable x. In this manner, the original value is acquired, without the 9 0s added.

We did not explain decompression at the beginning of this chapter because the first step is to understand compression. We yet cannot wrap our heads around the compression and the decompression.

Checking over 160 Million Key Value Pairs

```
ch2416.py
import leveldb
import subprocess
import json
display= 0
def perror(s , hash , finalstr , hasht , a , ripemd , offset , nCode , amt , nHeight , status , out , outarr , cnt) :
    print "Offset %s nCode %d:%s amt %s nHeight %d" % (offset , nCode , bin(nCode) , amt , nHeight)
    print "Out %d outarr %s" % (out , outarr)
    print "%s:%d" % (s , hasht)
```

```

    print "Hash:%s" % hash.encode('hex')
    print "XORBytes:%s" % finalstr.encode('hex')
    print "bitcoin-cli %s" % a['scriptPubKey']['hex'][6:-4]
    print "ripmed %s" % ripemd.encode('hex')
    print "a=%s" % a
    if status == 0:
        exit(0)
    print "cnt is %d" % cnt
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def print1(s):
    if display == 1:
        print s
def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)
def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1

```

```
    n = (n << 7) | (chData & 0x7F)
    if chData & 0x80 == 128:
        n = n + 1
    else:
        return (n,offset)
def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
        if k == 8:
            k = 0
    return finalstr
def bitson(num , times):
    times = times * 8
    times = times + 1
    arr = []
    if num & 1 == 1:
        arr.append(1 + times)
    if num & 2 == 2:
        arr.append(2 + times)
    if num & 4 == 4:
        arr.append(3 + times)
    if num & 8 == 8:
        arr.append(4 + times)
    if num & 16 == 16:
        arr.append(5 + times)
    if num & 32 == 32:
        arr.append(6 + times)
    if num & 64 == 64:
        arr.append(7 + times)
    if num & 128 == 128:
        arr.append(8 + times)
    if num == 0:
        arr.append(0 + 2 + times)
    return arr
cntspent = cntnon = cnttbd = hash1 = hash2 = hash3 = hash4 = hash5 = hash6 = hash7 = hash8 = 0
nHeight = 0
a = ''
amt = 0
hasht = 0
ripemd = ''
```

```

def displaychainstate(finalstr , hash):
    global cnt1 , arrayi , cntspent , cntnon , cnttbd , nHeight , a , amt , hasht , display , ripemd
    global hash1 , hash2 , hash3 , hash4 , hash5 , hash6 , hash7 , hash8
    print1(hash.encode('hex'))
    finalstr = finalstr.decode('hex')
    (version , offset) = base128(finalstr, 0)
    (nCode, offset) = base128(finalstr, offset)
    hash = hash[::-1]
    outarr = []
    if nCode == 0:
    if 0 not in outarr:
    outarr.append(0)
    if nCode & 0x01 == 1:
    outarr.append(0)
    if nCode & 0x02 == 2:
    if 0 not in outarr:
    outarr.append(0)
    if nCode & 0x04 == 4:
    outarr.append(1)
    increase = 0
    if nCode >= 8:
    print1( "———nCode %d:%s:%d" % (nCode , bin(nCode) , nCode >> 3))
    print1( "hash %s" % hash.encode('hex'))
    print1( "bytes %s" % finalstr.encode('hex'))
    if (nCode & 0x02 == 0) and (nCode & 0x04 == 0):
    increase = 1
    cntvout = (nCode >> 3) + increase
    print1( "cntvout %d offset %d increase %d" % (cntvout,offset , increase))
    print1( "bytes %s" % finalstr.encode('hex'))
    for i in range ( 0 , cntvout) :
    hexvout = ord(finalstr[offset:offset + 1])
    while hexvout == 0:
    offset = offset + 1
    hexvout = ord(finalstr[offset:offset + 1])
    if hexvout != 0:
    break
    print1( "hexvout %x" % hexvout)
    arrbits = bitson(hexvout , i)
    offset = offset + 1
    for bits in arrbits:
    outarr.append(bits)

    print1( "outarr %s" % outarr)
    for out in outarr:
    print1( "out %d offset %d" % (out,offset))
    outs = "%s" % out
    raw = subprocess.check_output(["bitcoin-cli" , "gettxout" , hash.encode('hex') , outs])

```

```
if len(raw) == 0:
    cntspent = cntspent + 1
    return
a = json.loads(raw)
print1( "a=%s" % a)
typescr = a['scriptPubKey']['type']
(amt , offset) = base128a(finalstr, offset )
hasht = ord(finalstr[offset:offset+1])
if hasht == 0x80:
    hasht = 0x80 + ord(finalstr[offset + 1:offset+2])
    offset = offset + 1
if a['scriptPubKey']['type'] == 'nonstandard':
    cntnon = cntnon + 1
    return
elif hasht == 0 and typescr == 'pubkeyhash' :
    hash1 = hash1 + 1
    print1( "A1 pubkeyhash hasht == 0")
    ripemdoffset = 21
    ripemd = finalstr[offset + 1 : offset + ripemdoffset]
    addrb = a['scriptPubKey']['hex'][6:-4] == ripemd.encode('hex')
    if addrb == False:
        perror("A1 pubkeyhash hasht == 0" , hash , finalstr , hasht , a , ripemd , offset , nCode ,amt , nHeight ,
        1 , out , outarr , cnt)
    else:
        print1( "=====A1 pubkeyhash passed")
        elif hasht > 0 and typescr == 'pubkeyhash' :
            hash2 = hash2 + 1
            print1( "A2 pubkeyhash hasht > 0")
            ripemdoffset = 27
            ripemd = finalstr[offset + 1 : offset + ripemdoffset]
            addrb = a['scriptPubKey']['hex'] == ripemd.encode('hex')
            if addrb == False:
                perror("A2 pubkeyhash hasht > 0" , hash , finalstr , hasht , a , ripemd , offset , nCode ,amt , nHeight ,1,
                out , outarr , cnt)
            else:
                print1( "====A2 pubkeyhash passed")
                elif hasht == 1 and typescr == 'scripthash':
                    hash3 = hash3 + 1
                    print1( "A3 scripthash hasht == 1")
                    ripemdoffset = 21
                    ripemd = finalstr[offset + 1 : offset + ripemdoffset]
                    addrb = a['scriptPubKey']['hex'][4:-2] == ripemd.encode('hex')
                    if addrb == False:
                        perror("A3 scripthash hasht == 1" , hash , finalstr , hasht , a , ripemd , offset , nCode ,amt , nHeight ,1,
                        out , outarr , cnt)
                    else:
```



```

print1( "=====A3 scripthash passed")
elif (hasht == 2 or hasht == 3) and typescr == 'pubkey':
    hash4 = hash4 + 1
    print1( "A4 pubkey hasht == 2 or 3")
    ripemdoffset = 33
    ripemd = finalstr[offset + 1 : offset + ripemdoffset]
    addrb = a['scriptPubKey']['hex'][4:-2] == ripemd.encode('hex')
    if addrb == False:
        perror("A4 pubkey hasht == 2 or 3", hash, finalstr, hasht, a, ripemd, offset, nCode, amt, nHeight, 1,
            out, outarr, cnt)
    else:
        print1( "=====A4 pubkey passed")
        elif (hasht == 4 or hasht == 5) and typescr == 'pubkey':
            hash5 = hash5 + 1
            print1( "A5 pubkeyhash hasht == 4 or 5")
            ripemdoffset = 33
            ripemd = finalstr[offset + 1 : offset + ripemdoffset]
            addrb = a['scriptPubKey']['hex'][4:68] == ripemd.encode('hex')
            if addrb == False:
                perror("A5 pubkey hasht == 4 or 5", hash, finalstr, hasht, a, ripemd, offset, nCode, amt, nHeight, 1,
                    out, outarr, cnt)
            else:
                print1( "=====A5 pubkey passed")
                elif typescr == 'pubkey' and hasht >= 6:
                    hash6 = hash6 + 1
                    print1( "A6 pubkey hasht >= 6")
                    ripemdoffset = ord(finalstr[offset + 1]) + 3
                    ripemd = finalstr[offset + 1 : offset + ripemdoffset]
                    addrb = a['scriptPubKey']['hex'] == ripemd.encode('hex')
                    if addrb == False:
                        perror("A4 pubkey hasht >=6", hash, finalstr, hasht, a, ripemd, offset, nCode, amt, nHeight, 1,
                            out, outarr, cnt)
                    else:
                        print1( "=====A6 pubkey passed")
                        elif typescr == 'multisig' and hasht <= 255:
                            hash7 = hash7 + 1
                            print1( "A7 multisig hasht <= 255")
                            ripemdoffset = hasht
                            soffset = offset
                            ripemd = finalstr[offset + 1 : offset + ripemdoffset]
                            offset = 1
                            while (True):
                                print1( "%x" % ord(ripemd[offset]))
                                if ord(ripemd[offset]) == 0x21 or ord(ripemd[offset]) == 0x41 or ord(ripemd[offset]) == 0x36:
                                    offset = offset + 0x1 + ord(ripemd[offset])
                                else:

```

```
break
offset = offset + 2
addrb = a['scriptPubKey']['hex'] == ripemd[0:offset].encode('hex')
offset = soffset + offset
ripemdoffset = 1
if addrb == False:
    perror("A7 multisig hasht <= 255", hash, finalstr, hasht, a, ripemd, offset, nCode, amt, nHeight, 1,
    out, outarr, cnt)
else:
    print1("=====A7 multisig passed")
else:
    hash8 = hash8 + 1
    ripemdoffset = hasht
    perror("else", hash, finalstr, hasht, a, ripemd, offset, nCode, amt, nHeight, 0, out, outarr, cnt)
    offset = offset + ripemdoffset
    (nHeight, offset) = base128(finalstr, offset)

tmp = a['value'] * 100000000
amountb = tmp - float(amt) < 0.001
if amountb == False:
    perror("Amount Issue", hash, finalstr, hasht, a, ripemd, offset, nCode, amt, nHeight, 1, out,
    outarr, cnt)

nHeights = "%s" % nHeight
raw1 = subprocess.check_output(["bitcoin-cli", "getblockhash", nHeights])
if len(raw1) == 0:
    perror("Height issue", hash, finalstr, hasht, a, ripemd, offset, nCode, amt, nHeight, 1, out,
    outarr, cnt)
raw2 = subprocess.check_output(["bitcoin-cli", "getblock", raw1])
c = json.loads(raw2)
heightb = nHeight == c['height']
if heightb == False:
    print "The original height %d bitcoin-cli %d" % (nHeight, c['height'])
else:
    unicodelist = c['tx']
    asciilist = map(str, unicodelist)
    tranb = hash.encode('hex') in asciilist
    if tranb == False:
        print "The Transaction is not found %s" % (hash.encode('hex'))

xorkey = ""
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstate13")
for k,v in db.Rangelter():
    cnt = cnt + 1
    if cnt % 100000 == 0:
        print "cnt=%d cntnon=%d cntspent=%d cnttbd=%d" % (cnt, cntnon, cntspent, cnttbd)
    if k[:1] == 'c':
```

```

if xorkey == '':
    finalstr = v.encode('hex')
else:
    finalstr = xorstring(v, xorkey)
    hash = k[1:]
    displaychainstate(finalstr, hash)
    elif k[:1] == 'B':
        pass
    elif ord(k[:1]) == 14:
        print "xoring key found"
        xorkey = v[1:]
    else:
        print "Unknown found %s:%s" % (k.encode('hex'), v.encode('hex'))
print hash1, hash2, hash3, hash4, hash5, hash6, hash7, hash8

```

Output

```

cnt=16300000 cntnon=168 cntspent=2216436 cnttbd=0
22873682 1 4,493,354 7653 36623 446 524758 0

```

The pubkeyhash type has over 22 million key-value pairs. This program took over a week to run. We were happy was that the last variable hash8 had a value of 0 which means that all possible types were accounted for.

The reading of the UTXO set is extremely fast. What is very slow is confirming the parsing or understanding of the UTXO set. Starting a new process in Python is very slow and there is absolutely nothing we can do about it.

Running our code on the entire UTXO set takes around 6 days. It requires a separate machine whose only job is to run code unattended.

```
caffeinated -i python ch2416.py
```

The program caffeinate has been a life saver. Normally when the Mac goes to sleep, all code running on it, also sleeps. However, by running the code through caffeinate, the Mac sleeps but the code still runs. All code is put into one large program.

The output shows the number of different hash types found in our UTXO set. By far, the most common hash types are the pubkeyhash and the scripthash.

The function called perror displays the entire state of a key-value pair only when an error occurs. It saves time in writing separate if statements for every new hash type. The parameter called status decides if the program should quit out when an error occurs. Many a times, errors take place and we must quit out and at times the code should continue processing. After all, we are dealing with millions of leveldb records.

The print1 function gets called only if something is to be displayed. It is an intelligent print. The display variable is used to selectively display print statements. When the value of the display variable is 1, then and only then, will the original print statement be called. This approach is useful for debugging, especially when dealing with millions of key-value pairs.

CHAPTER 25

Wallets

In this chapter, we will learn the structure of a Bitcoin wallet and its data storage patterns. A Bitcoin wallet is a storage of key value pairs. At one level, a Bitcoin wallet is nothing but a file called wallet.dat. This file is stored in the main Bitcoin folder.

The problem with storing inputs and outputs in a blockchain is that there are no running totals of Bitcoins owned by individual Bitcoin addresses. So, there is no definite count on the Bitcoins that can be spent. The Bitcoin blockchain is simply data stored in blk.dat files. They are large and bulky, so it is unfeasible to scan them for individual balances, when required for specific Bitcoin addresses.

This is where a wallet comes in. A wallet stores Bitcoins and Bitcoin related information. It stores all your Bitcoin addresses. As a matter of fact, the Bitcoin addresses are created by the wallet itself. It stores extra data also with every Bitcoin address, like the private key that created this Bitcoin address, etc.

You may have noticed that we stopped short of calling leveldb, the best key-value storage database. For reasons still not known to us, the Bitcoin developers chose another storage that goes by the name of Berkley-DB, to store all these key value pairs.

So, let's install Berkeley-DB version 4 and not version 6, which is the current version due to some copyright issues. The commands on our Mac OSX is as follows.

```
$ brew install berkeley-db4
```

Then install the python module using the following command:

```
$ sudo pip install bsddb3
```

In case you get an error, which happened to us, give the command as

```
$ sudo BERKELEYDB_DIR=$(brew --cellar)/berkeley-db4/4.8.30 pip install bsddb3
```

Google plays an important role here as it provides information, explains these errors and helps rectify them. We have no choice but to spend a lot of our time installing software because they are made up of sugar and spice.

Next, copy the wallet file, wallet.dat in the folder where the python code is written.

Counting the Number of Key-Pairs in the Berkley DB wallet.dat File

```
ch2501.py
from bsddb3 import db
cnt = 0
wallet = db.DB()
wallet.open('./wallet.dat', "main")
```

```

cursor = wallet.cursor()
rec = cursor.first()
while rec:
    key = rec[0]
    value = rec[1]
    rec = cursor.next()
    cnt = cnt + 1

print "Number of key values pairs in wallet %d" % cnt

```

Output

Number of key values pairs in wallet 611

A caveat here: This is our wallet, it stores the Bitcoins we own. Therefore, the output we get, will never match yours. Your Bitcoin wallet will store your money. One of the reasons we included our wallet file along with the code is because there are no bitcoins in this wallet anymore so, nothing to steal. We also assume that our wallet version is Bitcoin Core 0.14 or less.

There is no way you can duplicate the output shown here unless you use our wallet file. This is not like block 0 of the Bitcoin blockchain, which is the same for all mankind. So please use this wallet.dat file that we have made public along with the source code, it will be easier to follow code. There are in fact multiple wallet.dat files.

The programs in the chapter attempt to explain the key value pairs that make up a wallet and what they represent, so the output is brushed aside.

We shut down the bitcoind daemon and delete wallet.dat file present in the Bitcoin folder. Now, start the bitcoind application with the -printtoconsole option. We assumed that the bitcoin daemon would throw some exceptions or give errors as there was no wallet. Well it does not, it creates a fresh new empty wallet.

Given below are some of the wallet related output:

Output

```

2017-01-13 05:42:43 Keys: 0 plaintext, 0 encrypted, 0 w/ metadata, 0 total
2017-01-13 05:42:43 Performing wallet upgrade to 60000
2017-01-13 05:42:43 keypool added key 1, size=1

2017-01-13 05:42:47 keypool added key 100, size=100
2017-01-13 05:42:47 keypool added key 101, size=101
2017-01-13 05:42:48 keypool reserve 1
2017-01-13 05:42:48 keypool keep 1

setKeyPool.size() = 100

```

In other words, the daemon creates some 100 keys. As of now, all this is Greek and Latin to us. For every program, we may show you two outputs, one with a wallet holding some transactions and one with an empty wallet.

The output displayed with an empty wallet will match the output you see. We call this empty file, wallete.dat. So, should you. Replace the file wallet.dat with wallete.dat.

The output of the very first program with wallete.dat.

Output

Number of key values pairs in wallet 312

The number of keys have almost dropped by half. Even though they are key value pairs, databases leveldb and Berkley-DB are as different as chalk and cheese. At the end of the chapter, we will look at the changes made with Bitcoin Core version 0.15.

Back to the program code.

There is the usual import or from statement. Then a Berkley-DB handle called wallet is created. The DB function returns this opaque handle. This is the first function called in any Berkley-Db code.

The handle is then used to open a file using the Open function. The first parameter is the full path name of the physical file, wallet.dat or wallete.dat. The next is the name of the database, as a Berkley-DB file can have multiple databases in one physical file to complicate the issue. The Bitcoin folks called the database main, so it is used as is.

Now, to read the data, first a cursor is created and it is named cursor in our code. Then the first key-value pair is read into the tuple called rec using the function first.

There is a while loop to read the data in the tuple. The first member of the tuple rec[0] has the key and the second member rec[1] has the value of this key. To retrieve the second key-value pair, the next function of the cursor is used.

This approach helps scan all the key value pairs in the wallet.dat file. There are 611 key value pairs that make up our wallet. You may have a different count.

Starting with a wallet key called tx

```
ch2502.py
from bsddb3 import db
cnt = 0
wallet = db.DB()
wallet.open('./wallet.dat', "main")
cursor = wallet.cursor()
rec = cursor.first()
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:3] == "tx":
        cnt = cnt + 1
        if cnt == 1:
            print ord(key[0:1])
        rec = cursor.next()

print "Number of Transaction key values pairs %d" % cnt
```

Output

2

Number of Transaction key values pairs 280

Output with wallete.dat

Number of Transaction key values pairs 0

The keys in a wallet have a unique format. They start with a number that gives a count on the number of bytes taken up by the key name. Most string formats use an initial length byte to store the length of the string.

The tx key represents a transaction. It starts with a 2 for the length of the name tx.

In the wallet, there are 280 or nearly half of the key value pairs that belong to the tx key. It is the most popular key value pair in a wallet.

```
bitcoin-cli getaddressesbyaccount ""
[
  "1KGCWjm3ucJ1Vqfi7sKwYekuyJQjVVjSN",
  "159bpitnACnYo5YKXpfN3jrRsDKSSxo2sQ",
  "18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7",
  "1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74",
  "1CYtDq8VYcb4CqaFysYgtKa15MCi1iWc7",
  "1DTGvGaFHfeHvbZThZrPbddM4KgF6E4QHn"
]
```

Before executing the next program, let's look at the Bitcoin addresses in our wallet. We created the above 6 Bitcoin addresses in the past. It is the wallet that creates a new Bitcoin address, stores it and uses it.

The command `getaddressesbyaccount` obviously reads some keys in the wallet file `wallet.dat`. If things do not work as advertised then the problem is with the empty account name string.

Now shutdown the Bitcoin server and rename `wallet.dat` to `walletold.dat`. Then rename the `wallet.dat` file to the original `wallet.dat` file.

Our wallet is brand new so no Bitcoin addresses in this wallet have received/sent any money. The wallet code seems to be working as there is no transaction output referencing our Bitcoin address in it.

But what is our Bitcoin address?

```
bitcoin-cli getaddressesbyaccount ""
[
  "192VaoqRQhYyjSp7hXYWfjpKtvG42XQYW8"
]
```

Our Bitcoin address starts with 192V, your Bitcoin address will be different. Now it's time to send some Bitcoins to this address.

```
AddToWallet
7d7e485792e7b57fd286d0c015d517352615a41da30edbe305791b044c9b171d new
```

The output of the Bitcoin daemon shows the above line. We enter this transaction hash value starting with 7d7 in our blockchain browser. Surprisingly, this transaction hash is of the transaction that sends money to our newly minted Bitcoin address, starting from 192V. You will not see this line, we added a `printf` in the source code.

Run the above program again, the count/number of key value pairs for the tx key is still 0.

That's because the wallet file is an old one. When the new `wallet.dat` is copied in the Bitcoin folder to the current folder, the output is as follows:

```
Output
2
Number of Transaction key values pairs 1
```

Things are looking good now and hopefully you will get a similar output. Your transaction hash will be different.

The new wallet has a different Bitcoin address now.

```
bitcoin-cli getaddressesbyaccount ""  
[  
    "1LYtAevH7Uz4GfQu7FgzkNmeEfNqA5uqdg"  
]
```

At times, the output will be from three different wallets.

Decoding the Wallet Key tx

```
ch2503.py  
from bsddb3 import db  
import subprocess  
def htx(tx):  
    rawtx1 = subprocess.check_output(["bitcoin-cli", "decoderawtransaction", tx[:382]])  
    print rawtx1  
wallet = db.DB()  
wallet.open('./wallet.dat', "main")  
cursor = wallet.cursor()  
rec = cursor.first()  
cnt = 0  
while rec:  
    key = rec[0]  
    value = rec[1]  
    if key[1:3] == "tx":  
        if cnt == 0 or cnt == 1:  
            print "%d %c%c Length %d" % (ord(key[0]), ord(key[1]), ord(key[2]), len(key))  
            print "Hash %s" % key[3:][::-1].encode('hex')  
            htx(key[3:][::-1].encode('hex'))  
        else:  
            exit()  
        cnt = cnt + 1  
        rec = cursor.next()
```

Output

2 tx Length 35

Hash d4c48d3ca67d614d27ac86ed1bff4d3d29213454d56c78af20735040626f8a00

```
{  
    "txid":  
    "d4c48d3ca67d614d27ac86ed1bff4d3d29213454d56c78af20735040626f8a00",  
    "hash":  
    "d4c48d3ca67d614d27ac86ed1bff4d3d29213454d56c78af20735040626f8a00",  
    "size": 191,  
    "vsize": 191,  
    "version": 1,  
    "locktime": 0,  
    "vin": [  
        {  
            "txid":
```



```

    "5279ca4eddb7612ea1d270fcfe8b82098102dba09699e21e4696616a887acf5",
    "vout": 3,
    "scriptSig": {
      "asm":
        "30440220313285cb56b7dd7761e5bdeeb535bf7f4829f04b557a7279164f09fbd3df81fc
        022075d94fc33b2c5c0541c41f2493fb37549fb943e4107be0d168efe41205579d8e[ALL]
        028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b",
      "hex":
        "4730440220313285cb56b7dd7761e5bdeeb535bf7f4829f04b557a7279164f09fbd3df81
        fc022075d94fc33b2c5c0541c41f2493fb37549fb943e4107be0d168efe41205579d8e012
        1028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b"
    },
    "sequence": 4294967295
  },
  ],
  "vout": [
    {
      "value": 0.00055000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 c847b6d84ecf8048474b19c4a2330409ff32a1ad
        OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914c847b6d84ecf8048474b19c4a2330409ff32a1ad88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1KFz4ACsUqeFrJsFD2P3YgU7zS4WESootv"
        ]
      }
    }
  ]
}

```

Let's understand the tx key and all its gory details. It starts with the number 2, which indicates that the key name has a length of 2. These are the character's t and x. The size of the entire key is 35 bytes of which 32 bytes are for the transaction hash. This is because $35 - 3 = 32$.

Whose transaction hash you will ask?

Each time Bitcoins are spent or received, the wallet store will hold the transaction ids of these transactions. The 276 tx keys contain two types of transactions. The first type is where one of the above 6 Bitcoin address are mentioned in an output.

This type of transaction refers to the incoming/received Bitcoins which can be spent later. The other type of transaction that interests our wallet is the output, where the Bitcoins are spent, though here our Bitcoin address will be mentioned indirectly.

The task of the Bitcoin server code here is to read the input which will reveals the transaction id and output index of the Bitcoin which is spent in this transaction. The wallet code is part of the code that makes up Bitcoin core. Our code

receives every transaction that is mined in a block. It is this wallet code that decides whether a tx type key pair needs to be added to the wallet or not. It decides after looking at the output and as well as the input pointing to an output that is one of the Bitcoin addresses the wallet generated. If it is, a new tx key pair is added to the wallet.

The transaction hash of the first key starts with d4c48. The blockchain.info shows the transaction with one input and one output.

The input Bitcoin address is 18zg, which is one of the Bitcoin addresses created by the wallet in the past. It is the third address in the displayed list.

The output is also a well-known address ending by Sootv, our Zebpay Bitcoin address. This transaction shows that the Bitcoins were transferred from address 18zg to 1Kz4. We transferred 0.00055 BTC in this transaction.

Back to our code. The first check is on the name, tx in the second and third bytes. As abundant caution, the transaction hash is also displayed.

Then function htx is called with the hash of the tx key. These bytes are passed through the decoderawtransaction command, which gives an output similar to the one displayed by blockchain.info. This proves that the value field of the key tx is simply the raw bytes of a transaction hash.

In our code, we have specifically displayed only the first two key-value pairs. The second transaction hash value starts with d4ab.

This second key is also simple to understand as this time the Bitcoin address 1DTGv is the first of the two outputs that receives some Bitcoins, a very small value. This Bitcoin address is the last in the list. The second output is ignored.

Thus, it is safe to state that the 276 key-value pairs are the ids of those transactions where Bitcoins are received or spent.

It is easier to scan 280 key value pairs than scan nearly 100 GB of data stored in .dat files. Each time a block is written to disk, bitcoind or bitcoin-qt or the Bitcoin source will scan this newly minted block.

It would only check if the Bitcoin addresses (in our case 6, you may have 100), are present either indirectly in the input or directly in the output. Bitcoin address can only be present in an output; the input refers to them indirectly. So, either transfer some Bitcoins to someone or receive some Bitcoins from someone, recopy the wallet.dat file and see how the number of tx keys change.

In summary: The wallet dynamically stores every transaction that contains the Bitcoin addresses it has created. The transaction will be of two types, either directly present in the output or indirectly pointed to in the inputs.

Please switch between the two wallets by changing the wallet name in the program.

For some reason, version 0.15 gives us an error on method decoderawtransaction. The transaction bytes are by far too many so if we restrict the bytes to 382, everything works as advertised. We have no idea what the extra bytes are. We are too tired to investigate any further.

A tx Field is Part of the Outputs or Inputs?

```
ch2504.py
from bsddb3 import db
import subprocess
import json
import pybitcointools
def htx(tx, laddr):
    rawtx1 = subprocess.check_output(["bitcoin-cli", "decoderawtransaction", tx[:382]])
    rawtxd = json.loads(rawtx1)
```

```

vout = rawtxd["vout"]
match = 0
for vouto in vout:
    try:
        baddrlist = vouto['scriptPubKey']['addresses']
        for baddr in baddrlist :
            if baddr in lbaddr:
                print "Recieved Bitcoins"
                print "Output Address %s" % baddr
                match = 1
            except:
                continue
        if match == 0:
            vin = rawtxd["vin"]
            for vino in vin:
                txid = vino["txid"]
                output = vino["vout"]
                rjson = pybitcointools.fetchtx(txid)
                rawtx2 = subprocess.check_output(["bitcoin-cli" , "decoderawtransaction" , rjson])
                rawtx2 = json.loads(rawtx2)
                for baddvout in rawtx2["vout"][output]["scriptPubKey"]["addresses"]:
                    if baddvout in lbaddr:
                        print "Spent Bitcoins"
                        print "Output %s:%d" % (txid,output)
                        print "Address in Input %s" % baddvout
                        match = 1
                    if match != 1:
                        print "The sun rises from the west"
                        exit()

wallet = db.DB()
wallet.open('./wallet.dat' , "main" )
cursor = wallet.cursor()
rec = cursor.first()
cnt = 0
rawtx = subprocess.check_output(["bitcoin-cli" , "getaddressesbyaccount" , ""])
lbaddr = json.loads(rawtx)
print lbaddr
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:3] == "tx":
        if cnt == 0 or cnt == 1:
            print
            print "%d %c%c Length %d" % (ord(key[0]) , ord(key[1]) , ord(key[2]) , len(key))
            print "Hash %s" % key[3:][::-1].encode('hex')
            htx(value.encode('hex') , lbaddr)

```

```
else:
    exit()
    cnt = cnt + 1
    rec = cursor.next()
```

Output

```
[
    "1KGCWjm3ucJ1Vqfi7sKwYekuyJQjVVjSN",
    "159bpitnACnYo5YKXpfN3jrRsDKSSxo2sQ",
    "18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7",
    "1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74",
    "1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7",
    "1DTGvGaFHfeHvbZThZrPbddM4KgF6E4QHn"
]
[u'1KGCWjm3ucJ1Vqfi7sKwYekuyJQjVVjSN',
u'159bpitnACnYo5YKXpfN3jrRsDKSSxo2sQ',
u'18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7',
u'1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74',
u'1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7',
u'1DTGvGaFHfeHvbZThZrPbddM4KgF6E4QHn']
2 tx Length 35
Hash d4c48d3ca67d614d27ac86ed1bff4d3d29213454d56c78af20735040626f8a00
Spent Bitcoins
Output 5279ca4ededb7612ea1d270fcfe8b82098102dba09699e21e4696616a887acf5:3
Address in Input 18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7

2 tx Length 35
Hash d4aba94f74554222322d374571d7fdb430f344d6ccbc64b2cda12a84beaa801
Recieved Bitcoins
Output Address 1DTGvGaFHfeHvbZThZrPbddM4KgF6E4QHn
```

This program substantiates the fact that every tx key represents one transaction where a Bitcoin address receives Bitcoins or it has spent some of its own.

First, the `getaddressesbyaccount` command is executed. The output is then converted into a simple list, variable `lbaddr`, which now contains our 6 Bitcoin addresses.

The while loop of the previous program is used for the first tx key. You will handle the second key by yourself. The value field of the tx key and the Bitcoin addresses list, `lbaddr` is given to a function called `htx`. This function has plenty of code.

The value field `tx` has the raw transaction bytes. The `decoderawtransaction` method is given the raw transaction bytes and the returned string is then converted into a Python dictionary. The dictionary variable `rawtxd` has this transactions data as a dictionary.

The focus here is on the `vout` list variable that contains a list of transaction outputs. This variable `vout` is the `out` field of the dictionary, `rawtxd`. In a for loop, each member of this list is scanned. The variable `vouto` represents each individual output. Thereafter, the `scriptPubKey` and addresses of the field are acquired.

The `baddrlist` variable is a list. The `baddr` variable accesses each member of this list. This address is checked with the list of Bitcoin addresses present in the `lbaddr` list. The `in` keyword is used in the loop.

When the if statement results in true, it means a match is found. Which means that the transaction outputs have a Bitcoin address matching an address in the lbaddr list. The match variable is set to 1 signifying match found. This means that our Bitcoin address has received Bitcoins.

Let's now consider a possibility that this transaction is the one where Bitcoins are being spent. The match variable must be 0 as the Bitcoin address cannot be present in the outputs, thus discontinuing the execution of the rest of the code. The focus will be on the inputs. As there are multiple inputs, the vin key is used to access a list of multiple inputs. The list variable vin has the value of the field.

In a for statement, each input in this list, vin is accessed. The transaction id and the output index number are acquired using the keys, txid and vout. Since it is time consuming to scan each transaction, the fetchtx function from the pybitcointools module is used. This function goes to the website blockchain.info and returns all the transaction raw bytes. Thereafter, the same old decoderawtransaction command decodes these transaction bytes. The returned output is converted into a dictionary.

A transaction has many outputs, so the output variable is used to retrieve our specific output. The output variable is the index in the list of outputs of the transaction whose hash is stored in the txid variable. The list of addresses is acquired using the keys, scriptPubKey and the field addresses in this dictionary. Then using the same in keyword, we ensure that one of the Bitcoin addresses exists in the outputs.

In other words, the lbaddr list has Bitcoin addresses and the baddvout list has the Bitcoin address found indirectly in the output.

The code is like the earlier one, the only difference is that the ids of different transaction outputs are read.

A word of caution: Whenever a key is read, please check if the key exists or an exception will be thrown. The Python purists insists on using the get function of a dictionary, but we prefer a try/catch.

The match variable is also initialized to 1. If its value is 0, then it signifies a major error as there is a transaction in the wallet that does not give/receive Bitcoins from our list of addresses. We check if the Bitcoin address is present in the transaction's output. If yes all is well. If no, there is one more check on the transaction, whether it has inputs is pointing to an output that has the Bitcoin addresses the wallet created. One of these must be true.

Finally remove the if, cnt, and else statements and run the program through the entire transactions keys, you will see no errors whatsoever. This confirms that the wallet code runs as advertised.

The new output looks like

Output

```
[u'192VaoqRQhYyjSp7hXYWfjpKtvG42XQYW8']
```

```
2 tx Length 35
```

```
Hash 7d7e485792e7b57fd286d0c015d517352615a41da30edbe305791b044c9b171d
```

```
Recieved Bitcoins
```

```
Output Address 192VaoqRQhYyjSp7hXYWfjpKtvG42XQYW8
```

Only one Bitcoin address starting with 192V. Some money is received so the second lot of code that reads the inputs will be ignored. The Bitcoin address starting with 192V is proudly shown.

The Defaultkey Dey

```
ch2505.py
from bsddb3 import db
import subprocess
```

```
import json
def hdefault(value , lbaddr):
    match = 0
    for baddr in lbaddr:
        rawpub = subprocess.check_output(["bitcoin-cli" , "validateaddress" , baddr])
        rawpub = json.loads(rawpub)
        pub = rawpub["pubkey"]
        print pub
        if pub == value[2:]:
            print "Default Key %s Matches Bitcoin Address %s" % (value[2:],baddr)
            match = 1
        if match == 0:
            print "No Match for Bitcoin address"

wallet = db.DB()
wallet.open('./wallet.dat' , "main" )
cursor = wallet.cursor()
rec = cursor.first()
rawtx = subprocess.check_output(["bitcoin-cli" , "getaddressesbyaccount" , ""])
lbaddr = json.loads(rawtx)
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:11] == "defaultkey":
        print "key %d:%s Length %d" % (ord(key[0]) , key[1:] , key.encode('hex') , len(key))
        print "value %s" % value.encode('hex')
        hdefault(value.encode('hex') , lbaddr)
    rec = cursor.next()
```

Output

```
key 10:defaultkey:0a64656661756c746b6579 Length 11
value 2103fcecc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596
027f3fc87f0605e725272b757b0c86edb9833ffd8547ab67ac7f29d11ef5f3e9fb
035ca61ec616a18cfcf2b5b631bd121b33429dc1fd7a999ec1b1069d2872e270e3
028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b
035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee
03fcecc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596
Default Key
03fcecc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596
Matches Bitcoin Address 1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7
029b80fb7f3066b94ded269c7898d1c6ad0a23612d0eca493826e1d81e3ae64268
```

For the next couple of programs, we will follow a template wherein we will take one key at a time and call a function to decode it value field. The key is read in a variable called key and the value in a variable called value.

This program decodes a key called defaultkey. The key should have been called defaultpublickey in our opinion. The value of this key unlike the tx key, has no other data associated with the key. The key is just a key.

The value field of the key starts with 21, the length of a public key. The 03 byte value confirms what follows is a public key. The function `hdefault` is passed this key value and a list of Bitcoin address.

Every address in the list of the Bitcoin addresses, `laddr` is passed through the `validateaddress` command.

```
bitcoin-cli validateaddress 1CYtDq8VYcb4CqaFfysYgtKa15MCi
{
  "isvalid": true,
  "address": "1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7",
  "scriptPubKey": "76a9147eb109268f079ce35848a104ad13938e85716b1b88ac",
  "ismine": true,
  "iswatchonly": false,
  "isscript": false,
  "pubkey": "03fcec0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596",
  "iscompressed": true,
  "account": ""
}
```

The `validateaddress` command as learnt earlier, takes a valid Bitcoin address and reveals a lot of information about it. We are interested in a key called `pubkey` as it has the public key that created this Bitcoin address.

The public key however, does not start with the length of the key as it is stored in the value field of the `defaultkey` key. So, the public key returned by the `validateaddress` command is compared with all but the first byte of the `defaultkey` key's value. If there is a match, the Bitcoin address is displayed and the variable `match` is set to 1. Here, the public key is the default key.

If the variable `match` is not 1, then there is a serious error and error message is displayed.

The Bitcoin address that matches in our case starts with 1CYt. You will also see a match but with a different Bitcoin address, but there must be a match.

The new output looks like

```
bitcoin-cli validateaddress 192VaoqRQhYyjSp7hXYWfjpKtvG42XQYW8
{
  "isvalid": true,
  "address": "192VaoqRQhYyjSp7hXYWfjpKtvG42XQYW8",
  "scriptPubKey": "76a91458094832ce744f5d8750e2c0ed73818b77ab7b8288ac",
  "ismine": true,
  "iswatchonly": false,
  "isscript": false,
  "pubkey": "0244e8644cb593375de665861ac95ed28867d242f4d3f701b30ec0643a596dec3",
  "iscompressed": true,
  "account": "",
  "hdkeypath": "m/0'/0'/0'",
  "hdmasterkeyid": "5302e0ff49c0ffcb4c391c4070b0a7e5f1ecc788"
```

Output

```
key 10:defaultkey:0a64656661756c746b6579 Length 11
value 210244e8644cb593375de665861ac95ed28867d242f4d3f701b30ec0643a596dec3
0244e8644cb593375de665861ac95ed28867d242f4d3f701b30ec0643a596dec3
```

Default Key
0244e8644cb593375de665861ac95ed28867d242f4d3f701b30ec0643a596decb3
Matches Bitcoin Address 192VaoqRQhYyjSp7hXYWfjpKtvG42XQYW8

This one is a no brainer. There is only one Bitcoin address, one public key which is the default one. The defaultkey key has a value field starting with 44e8, confirmed by the validateaddress command.

The Keymeta Key Occurs more than 100 Times

```
ch2506.py
from bsddb3 import db
import subprocess
import json
import time

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def hkeymeta(value):
    print "Value length %d" % (len(value))
    version = int(reversehash(value[0:8]),16)
    time1 = int(reversehash(value[8:16]) ,16)
    time2 = int(reversehash(value[8:24]) ,16)
    if time1 != time2:
        print "Error in time"
        print "Last 8 bytes %s" % value[16:24]
        print "Version %d %s" % ( version , time.ctime(time1))

wallet = db.DB()
wallet.open('./wallet.dat', "main" )
cursor = wallet.cursor()
rec = cursor.first()
rawtx = subprocess.check_output(["bitcoin-cli" , "getaddressesbyaccount" , ""])
lbaddr = json.loads(rawtx)
cnt = 0
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:8] == "keymeta":
        if cnt == 0 or cnt == 104 :
            print "key 1st byte %d Key:%s Length %d" % (ord(key[0]) , key[1:8] , len(key))
            print "Key Value %s" % (key[8:42].encode('hex'))
            hkeymeta(value.encode('hex') )
            print
            cnt = cnt + 1
            rec = cursor.next()
print "Total Number of keys %d" % cnt
```

Output

key 1st byte 7 Key:keymeta Length 42


```

Key Value 210200b36c9eb4535e21529b90d37385644fb419392d72b2442a8d42d11837b94f55
Value length 24
Last 8 bytes 00000000
Version 1 Sun May 1 08:15:35 2016

key 1st byte 7 Key:keymeta Length 42
Key Value 2103fcecc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596
Value length 24
Last 8 bytes 00000000
Version 1 Sun May 1 08:15:34 2016

Total Number of keys 105

```

There is a key called keymeta that occurs 105 times.

The program handles only two different values of the keymeta key. These are the first or 0th and then the last, 104th. Let's understand how the value of this key is structured. Your mileage will vary.

The first byte is 7, the length of the key name, keymeta. This is followed by 24 bytes of a public key, which start with the length 0x21 or 33 bytes plus the actual public key. The public keys normally start with a 02 or 03, as seen in the output.

The value is slightly more complex and its handled by the function hkeymeta. The field value is made up of 2 fields. Let's start with a version number that takes up 4 bytes or 8 characters. The int member converts the 4 bytes into a number which will be 1, for the time being.

The next field is time, basically the creation time of the public key. The time here is 8 bytes large or 16 characters. Unfortunately, the bytes in the string must be reversed before converting it into a number. The function that reverses a hash now reverses everything, including the string here. The time function then returns a readable time.

According to the source code, the version number has a constant value of 1. Therefore, the version field is always 1. Similarly, the time field is an 8-byte number. Therefore, the last 8 bytes are all 0's.

When the hash is reversed, the leading 0s do not change the value of a number, it simply appears larger to the human eye. Comparing the 8-byte time and the 4 byte times eventually is the same. Thus, variable time1 or time2, both can be used for the time value.

The keymeta key stores a list of public keys whose only goal in life is to create a Bitcoin address. This key gives information on when the public key was generated.

There are 105 public keys or keymeta keys, you may see more. The dates when the 0th and 104th public key was created are the same, thus implying that these 105 public keys were created at the same time.

The Pool Key

```

ch2507.py
from bsddb3 import db
import subprocess
import json
import time

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def hpool(value):

```

```
ver = int(reversehash(value[0:8]),16)
time1 = int(reversehash(value[8:24]), 16)
print "Ver %d %s %d" % (ver , time.ctime(time1) , len(value))
print "Public Key:%s" % (value[24:])

wallet = db.DB()
wallet.open('./wallet.dat', "main" )
cursor = wallet.cursor()
rec = cursor.first()
rawtx = subprocess.check_output(["bitcoin-cli" , "getaddressesbyaccount" , ""])
lbaddr = json.loads(rawtx)
cnt = 0
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:5] == "pool":
        if cnt == 0 or cnt == 1 or cnt >= 98:
            print "key 1st byte %d %s:%s Length %d" % (ord(key[0]) , key[1:5] , key[5:].encode('hex') , len(key))
            print "Index %d" % (int(reversehash(key[5:].encode('hex')),16))
            hpool( value.encode('hex') )
            cnt = cnt + 1
            rec = cursor.next()
print "Total Number of keys %d" % cnt
```

Output

```
key 1st byte 4 pool:0600000000000000 Length 13
Index 6
Ver 110200 Sun May 1 08:15:34 2016 92
Public Key:21027f3fc87f0605e725272b757b0c86edb9833ffd8547ab67ac7f29d11ef5f3e9fb
key 1st byte 4 pool:0700000000000000 Length 13
Index 7
Ver 110200 Sun May 1 08:15:34 2016 92
Public Key:2103b26b29957755989d0994088d5800c0ee5e0870e660480ac3896134fbc9e42b53
key 1st byte 4 pool:6800000000000000 Length 13
Index 104
Ver 120100 Thu Jun 23 20:28:33 2016 92
Public Key:21021a2f941cc8d1fed8de6d97c15ceed779547d1791bb150a1e4af3a3e340e99554
key 1st byte 4 pool:6900000000000000 Length 13
Index 105
Ver 120100 Sun Jul 17 22:30:31 2016 92
Public Key:21025d10566ad6649f6f6b46c9194173b51864d0ef684974a1601988728f859dabf3
Total Number of keys 100
```

This program explores the internals of a wallet, which will remain a mystery until you read the source code. For example, when a new wallet is created, it creates a set of 100 private keys. From these private keys, public keys are created and from these public keys, Bitcoin addresses are acquired.

A pool is a storage area for something to be used later. The 100 public keys are stored in 100 keys. There will always be a minimum of 100 keys in a pool.

The first byte of a key, as always is the size of the key name, pool. So, its 4. Then comes the name, pool. This uses up 5 bytes. The last 8 bytes out of the 13 bytes has a 64-byte number, 4 bytes each. In our case, it starts with 6 and ends with 105.

The if statement displays the first and second keys from the pool and all those with a count larger than 98. The only reason being is that the pool starts with 100 public keys and then when the topmost one gets used up to create a Bitcoin address, one key gets added at the bottom automatically. We will verify this in a short time.

The first pool key has an index value of 6, the next 7, the second last 104 and the last 105. These numbers are not chosen at random. They have some meaning.

The value field is always 92 bytes long. It starts with a version number 11.2 which changes to 12.01. Our current Bitcoin client version i.e. by running bitcoin-cli with no options is 12.01. Though we are running with the 0.13/0.14/0.15 series, our wallet has been created eons ago. With a change in the bitcoin core version, the version number in the pool value field also changes.

We are too laid back to create a Bitcoin address with the 0.13 series.

The version field takes up 8 bytes. Then we have the same 16-byte time, as in when the public key was created. The first one was created on the 1st of May 2016 and the last public key on the 17th of July, 2016.

So far, 24 bytes are accounted for. So, now 92 - 24 or 68 bytes are left out of 92 bytes. The public key starts with a length byte 21 that reveals a length of 66 bytes plus 2 for the length. This explains the size of 92 hex digits. As a reminder: we are dealing with an encoded string.

Let's work this out by hand also. The first public key waiting to be used is the one with an index 6. This is the public key.

21027f3fc87f0605e725272b757b0c86edb9833ffd8547ab67ac7f29d11ef5f3e9fb.

To create a new Bitcoin address, run the command below

```
bitcoin-cli getnewaddress
```

Given below is the result:

```
1KGCWjm3ucJ1Vqfi7sKwYekuyJQjVVjSN
```

To determine the public key that created this public key, execute the following command.

```
bitcoin-cli validateaddress 1KGCWjm3ucJ1Vqfi7sKwYekuyJQjVVjSN
{
  "isvalid": true,
  "address": "1KGCWjm3ucJ1Vqfi7sKwYekuyJQjVVjSN",
  "scriptPubKey": "76a91403742ca3f76a926ba61958cd4f4f417c8539023f88ac",
  "ismine": true,
  "iswatchonly": false,
  "isscript": false,
  "pubkey": "027f3fc87f0605e725272b757b0c86edb9833ffd8547ab67ac7f29d11ef5f3e9fb",
  "iscompressed": true,
  "account": ""
}
```

Aah! Indeed, a eureka moment! When we ask Bitcoin to create a new address, it does not really create a new address. It takes the first pool entry and simply hands over its details.

It also does more. Remember changes to our wallet will only be seen by our code when the bitcoin server is stopped as in:

```
bitcoin-cli stop
```

Copy the wallet.dat program back to the home folder and restart the bitcoin daemon.

Given below is the fresh output.

Output

```
key 1st byte 4 pool:0700000000000000 Length 13
Index 7
Ver 110200 Sun May 1 08:15:34 2016 92
Public Key:2103b26b29957755989d0994088d5800c0ee5e0870e660480ac3896134fbc9e42b53
key 1st byte 4 pool:0800000000000000 Length 13
Index 8
Ver 110200 Sun May 1 08:15:34 2016 92
Public Key:2103c2b8efe5c00ed11a34dbc6763e1ac81e5e61151980f736343bfad8363ed48f5b
key 1st byte 4 pool:6900000000000000 Length 13
Index 105
Ver 120100 Sun Jul 17 22:30:31 2016 92
Public Key:21025d10566ad6649f6f6b46c9194173b51864d0ef684974a1601988728f859dabf3
key 1st byte 4 pool:6a00000000000000 Length 13
Index 106
Ver 120100 Sun Jul 31 16:16:44 2016 92
Public Key:2103d7c75aef8c7f83670debf3370388f46cddc17aad42290aa0c60a930c11a7d022
Total Number of keys 100
```

We have substantiated our claims. The first index of the pool changes from 6 to 7. The pool index 6 simply disappears into a black hole. There are again 100 pool keys, as there will always be a constant 100 pool keys, come heaven or hell. However, the last pool entry is now 106 and not 105 earlier. Its date time stamp is 31st July 2016.

The output clearly shows that the newer key-value pairs take the version number of the bitcoin client we are running.

Your outputs will be different but the concepts remain the same. The pool entries are the Bitcoin addresses created in the past. We created 5 addresses earlier and then the sixth.

The first value of the index field was 6 and it is now 7. If we create one more bitcoin address, the first pool entry will start from 8. The new pool entries are added at the bottom with the new Bitcoin client version number and the new public key created. The version number changed to 12.01 The key word is always '100 pool entries'.

The new output is

Output

```
key 1st byte 4 pool:0200000000000000 Length 13
Index 2
Ver 130100 Fri Jan 13 11:12:43 2017 92
Public Key:2103363adec81c5712496e1488d60c743444aa25958e48ec4ac0231e7f66eab80278
key 1st byte 4 pool:0300000000000000 Length 13
Index 3
Ver 130100 Fri Jan 13 11:12:44 2017 92
```

```

Public Key:21036dbb30aea33c788fdaf29e9bd3a00c0ac710bac630933269710aa1b868a6e41d
key 1st byte 4 pool:6400000000000000 Length 13
Index 100
Ver 130100 Fri Jan 13 11:12:47 2017 92
Public Key:2103c0046c74babea434a61bbc0950a0607dff39e5bb011beec6c77fc8cea7536030
key 1st byte 4 pool:6500000000000000 Length 13
Index 101
Ver 130100 Fri Jan 13 11:12:47 2017 92
Public Key:2103f4718fc6e90e6dc030ac027b13c72d221d428cf489946984c47b388a8b23b294
Total Number of keys 100

```

The version number of the client software is not the 0.12 series but the 0.13 series. The index numbers start from 2 and not 1 and end at 101 and not 100. The years are 2017. The reasoning here is that 100 Bitcoin addresses are freely created when the wallet is first created. Please create one more bitcoin address and see how things change.

The Name Key

```

ch2508.py
from bsddb3 import db
import subprocess
import json

def hname(value ):
    print "Value %d len %d" % (ord(value) , len(value))

wallet = db.DB()
wallet.open('./wallet.dat', "main" )
cursor = wallet.cursor()
rec = cursor.first()
rawtx = subprocess.check_output(["bitcoin-cli" , "getaddressesbyaccount" , ""])
lbaddr = json.loads(rawtx)
cnt = 0
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:5] == "name":
        print "1st byte %d Key Name %s byte %s len %d" % (ord(key[0]) , key[1:5], ord(key[5:6]) , len(key))
        print "Bitcoin Address %s:%s " % (key[6:] , key[6:] in lbaddr)
        hname(value )
        cnt = cnt + 1
        rec = cursor.next()
print "Total Number of keys %d" % cnt

```

Output

```

1st byte 4 Key Name name byte 33 len 39
Bitcoin Address 1KGCWjm3ucJ1Vqfi7sKwYekuyJQjVVjSN:True
Value 0 len 1
1st byte 4 Key Name name byte 34 len 40
Bitcoin Address 159bpitnACnYo5YKXpfN3jrRsDKSSxo2sQ:True

```

```
Value 0 len 1
1st byte 4 Key Name name byte 34 len 40
Bitcoin Address 18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7:True
Value 0 len 1
1st byte 4 Key Name name byte 34 len 40
Bitcoin Address 1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74:True
Value 0 len 1
1st byte 4 Key Name name byte 34 len 40
Bitcoin Address 1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7:True
Value 0 len 1
1st byte 4 Key Name name byte 34 len 40
Bitcoin Address 1DTGvGaFHfeHvbZThZrPbddM4KgF6E4QHn:True
Value 0 len 1
Total Number of keys 6
```

The name key is very simple. Out of the total length of 39 or 40 bytes, the first byte is taken up for the length of the key, then comes the actual key called name, which is 4 bytes. Together, they take up 5 bytes.

The Bitcoin address can either be 33 or 34 bytes large. One byte is used to store the variable Bitcoin address size, which is followed by the Bitcoin address. Therefore, the key size is 39 or 40 bytes. The breakup is: 5 bytes for the key, 1 byte for length of Bitcoin address and then the address. The final length is either 39 or 40 bytes.

We check for the original source location of these addresses, though they are created by our wallet. At the beginning of this chapter, there were 6 Bitcoin addresses, so there are 6 key-value pairs for the name key. All the Bitcoin addresses satisfy the condition that the Bitcoin address found in the name variable must exist in the lbaddr list.

The value takes 1 byte and its value is 0. Since it is 0, we ignore it and the other values it can have. To summarize, for every Bitcoin address our wallet creates, there is a name key. The key again stores its Bitcoin address and nothing else.

The New Output is

```
Output
1st byte 4 Key Name name byte 34 len 40
Bitcoin Address 192VaoqRQhYyjSp7hXYWfjpKtvG42XQYW8:True
Value 0 len 1
Total Number of keys 1
```

One Bitcoin address, one output. No change.

A Key with more Purpose in Life

```
ch2509.py
from bsddb3 import db
import subprocess
import json

def hpurpose(value):
    print "Value 1st Byte %d %s Length %d" % (ord(key[0:1]) , value[1:] , len(value))

wallet = db.DB()
```

```

wallet.open('./wallet.dat', "main" )
cursor = wallet.cursor()
rec = cursor.first()
rawtx = subprocess.check_output(["bitcoin-cli", "getaddressesbyaccount", ""])
lbaddr = json.loads(rawtx)
cnt = 0
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:8] == "purpose":
        print "1st byte %d Key %s Len %d Bitcoin %s:%s Length %d" % (ord(key[0:1]), key[1:8], ord(key[8:9]),
        key[9:], key[9:] in lbaddr, len(key) )
        hpurpose(value)
        cnt = cnt + 1
        rec = cursor.next()
print "Total Number of keys %d" % cnt

```

Output

```

1st byte 7 Key purpose Len 33 Bitcoin 1KGCWjm3ucJ1Vqfi7sKwYekuyJQjVVjSN:True Length 42
Value 1st Byte 7 receive Length 8
1st byte 7 Key purpose Len 34 Bitcoin 159bpitnACnYo5YKXpfN3jrRsDKSSxo2sQ:True Length 43
Value 1st Byte 7 receive Length 8
1st byte 7 Key purpose Len 34 Bitcoin 18zg6FG5pu8Bpq73L54AYvB8phTw3qCCR7:True Length 43
Value 1st Byte 7 receive Length 8
1st byte 7 Key purpose Len 34 Bitcoin 1C6TH4Vf6nYUU6WhHyCTaLTJ9QE5fBAz74:True Length 43
Value 1st Byte 7 receive Length 8
1st byte 7 Key purpose Len 34 Bitcoin 1CYtDq8VYcb4CqaFfysYgtKa15MCi1iWc7:True Length 43
Value 1st Byte 7 receive Length 8
1st byte 7 Key purpose Len 34 Bitcoin 1DTGvGaFHfeHvbZThZrPbddM4KgF6E4QHn:True Length 43
Value 1st Byte 7 receive Length 8
Total Number of keys 6

```

The program shows another key called purpose, which is like the name key. There is one key for every Bitcoin address. The changes come in the value field. Other than that, the key's key field behaves like the name key.

It starts with a length of 7 which is the size of what follows, purpose. This value is always receive, which indicates that the purpose of this Bitcoin address is to receive Bitcoins.

The source code discloses that the purpose key name can either be send or receive or unknown or ". We are yet to come across a Bitcoin address with send as the value for the key purpose.

No new output as the purpose is receive.

The Version Key

```

ch2510.py
from bsddb3 import db

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

```

```
def hminversion(value):
    ver = int(reversehash(value) , 16)
    print "Min Version %d Length %d" % (ver , len(value))

def hversion(value):
    ver = int(reversehash(value) , 16)
    print "Version %d Length %d" % (ver , len(value))

wallet = db.DB()
wallet.open('./wallet.dat', "main" )
cursor = wallet.cursor()
rec = cursor.first()
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:8] == "version":
        print "Version Key 1st Byte %d name:%s len %d" % (ord(key[0:1]) , key[1:8] , len(key))
        hversion(value.encode('hex'))
    if key[1:11] == "minversion":
        print "Min Version Key 1st Byte %d name:%s len %d" % (ord(key[0:1]) , key[1:11] , len(key))
        hminversion(value.encode('hex'))
    rec = cursor.next()
```

Output

```
Version Key 1st Byte 7 name:version len 8
Version 130000 Length 8
Min Version Key 1st Byte 10 name:minversion len 11
Min Version 60000 Length 8
```

The version key reveals the bitcoin client version. The output shows the version number as 0.13.

There is also another singular key-value pair called minversion which is defined by the constant FEATURE_LATEST = 60000 in the source code. The value of the minversion key is 60000.

As always, reverse the string to get the number displayed.

The acc Key

```
ch2511.py
from bsddb3 import db
import subprocess
import json
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def hacc(value , lpubkey):
    print "Value %s" % value
    ver = int(reversehash(value[0:8]),16)
    print "Version %d PubKey %s" % (ver , value[8:] )
    print value[10:] in lpubkey
```



```

wallet = db.DB()
wallet.open('./wallet.dat', "main")
cursor = wallet.cursor()
rec = cursor.first()
rawtx = subprocess.check_output(["bitcoin-cli", "getaddressesbyaccount", ""])
lbaddr = json.loads(rawtx)
lpubkey = []
for baddr in lbaddr:
    raw1 = subprocess.check_output(["bitcoin-cli", "validateaddress", baddr ])
    raw1 = json.loads(raw1)
    print raw1['pubkey']
    lpubkey.append(raw1['pubkey'])
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:4] == "acc":
        print "Key 1st Byte %d name %s Length %d %s" % (ord(key[0:1]), key[1:4], len(key), key.encode('hex'))
        hacc(value.encode('hex'), lpubkey)
    rec = cursor.next()

```

Output

```

027f3fc87f0605e725272b757b0c86edb9833ffd8547ab67ac7f29d11ef5f3e9fb
035ca61ec616a18cfcf2b5b631bd121b33429dc1fd7a999ec1b1069d2872e270e3
028003e21c092ba3facaef44f92b3063389d345d03f54ac14933f7b4707e05b36b
035693f77e6c51ebdddec9ff00f40b8c3eb8a69250198efaf3d6522b0e80e789ee
03fcecc0fc2f33030f4bc081218671d899f883bd95c75d2a56772edb3d5662c596
029b80fb7f3066b94ded269c7898d1c6ad0a23612d0eca493826e1d81e3ae64268
Key 1st Byte 3 name acc Length 5 0361636300
Value 24d5010021035ca61ec616a18cfcf2b5b631bd121b33429dc1fd7a999ec1b1069d2872e270e3
Version 120100 PubKey 21035ca61ec616a18cfcf2b5b631bd121b33429dc1fd7a999ec1b1069d2872e270e3
True

```

The key called acc or account key name has nothing to do with an accounting entry. This key represents a public key. Prior to tackling the key, the list of Bitcoin addresses is assigned to the lbaddr list. Then, using the validateaddress command in a for loop, the public key associated with this Bitcoin address is obtained. The pubkey key is added to the list, lpubkey.

The key starts with the usual length, 3 and then the key name acc. For the first time, the key ends with a 00. Fortunately, there is nothing after the key name.

The value starts with the Bitcoin client version bytes, 12010000 and then from the 8th byte onwards is the public key. The public key starts with the length byte, 0x21 which is not present in the pubkey field.

Therefore, while looking for its entry in the list of public keys, as in lpubkey, the length of the public key is skipped and the check is from the 10th byte and not 8th. The logic for checking Bitcoin addresses used earlier is applied here on public keys as well. The return value is true as this public key matches one of the 6 public keys in our wallet or the list variable, lpubkey.

Once again: Our result confirms that the public key associated with the acc key matches a public key associated with a Bitcoin address created earlier in our wallet in the past.

The new output is

Output

0244e8644cb593375de665861ac95ed28867d242f4d3f701b30ec0643a596dec3

They simply knocked off, killed the acc key. No idea why !!!

The Bestblock Key

ch2512.py

```
from bsddb3 import db
import subprocess
import json
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def hbestblock(value):
    print "Version %d vHave %s length %d" % (int(reversehash(value[0:8]),16) , value[8:] , len(value))

wallet = db.DB()
wallet.open('./wallet.dat', "main")
cursor = wallet.cursor()
rec = cursor.first()
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:10] == "bestblock" and ord(key[0]) == 9:
        print "Key 1st Byte %d name %s len %d" % (ord(key[0:1]) , key[1:10] , len(key))
        hbestblock(value.encode('hex'))
    rec = cursor.next()
```

Output

Key 1st Byte 9 name bestblock len 10
Version 130000 vHave 00 length 10

The next key is called bestblock. The value field starts with the standard version value. According to the source code, there is an array of integers following the value field. The value is 0. We ignore all fields having a value of 0 as there is no way of verifying the field contents.

The new output is

Output

Key 1st Byte 9 name bestblock len 10
Version 130100 vHave 00 length 10

Other than the minor version no change, there is no significant change here.

The orderposnext Key

ch2513.py

```
from bsddb3 import db
```

```

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def horderposnext(value):
    print "Value %d Length %d" % (int(reversehash(value),16), len(value))

wallet = db.DB()
wallet.open('./wallet.dat', "main")
cursor = wallet.cursor()
rec = cursor.first()
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:13] == "orderposnext":
        print "Key 1st %d name %s len %d" % (ord(key[0:1]), key[1:13], len(key))
        horderposnext(value.encode('hex'))
        rec = cursor.next()

```

Output

```

Key 1st 12 name orderposnext len 13
Value 280 Length 16

```

At times, it is difficult to understand some keys. One of them is the key name called `orderposnext`. The key simply contains the key name and the value as an 8-byte integer.

The source code talks about the wallet being updated constantly but it has no connection with this value of 280. So, we simply throw our hands up in the air and walk away into the sunset.

The new output is

Output

```

Key 1st 12 name orderposnext len 13
Value 1 Length 16

```

The value is 1, one more reason to walk away.

The `bestblock_nomerkle` Key

```

ch2514.py
from bsddb3 import db

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def hbestblock_nomerkle(value):
    print "Version %d " % (int(reversehash(value[0:8]),16))
    print "CBlockLocator %s:%d" % (value[8:][0:100], len(value))

wallet = db.DB()
wallet.open('./wallet.dat', "main")
cursor = wallet.cursor()
rec = cursor.first()
while rec:

```

```
key = rec[0]
value = rec[1]
if key[1:19] == "bestblock_nomerkle":
    print "key 1st byte %d name %s len %d" % (ord(key[0:1]) , key[1:19] , len(key))
    hbestblock_nomerkle(value.encode('hex'))
    rec = cursor.next()
```

Output

```
key 1st byte 18 name bestblock_nomerkle len 19
Version 130000
CBlockLocator
1e26fec08078e76dfda186333f1aa46d3130f58516d0f12b000000000000000001727206
01c00a486244741b4b1d309aba8:1930
```

The key has a name bestblock_nomerkle. The value is a whopping 1930 bytes large. The code talks about a Set Best Chain mechanism, it is way beyond us to resolve this value so we leave it to you. Please take it seriously though. For now, we ignore this key.

Output

```
key 1st byte 18 name bestblock_nomerkle len 19
Version 130100
CBlockLocator
1e9ad129cfd3f3647cfb273962328f3161f668a982aad60020000000000000000b183c30
e60676ba5f1ec01f0550a25c295:1930
```

The Key Part of a Key called Key

```
ch2515.py
from bsddb3 import db

def hkey(value):
    print "Value %s Length %d" % (value[0:50] , len(value))

wallet = db.DB()
wallet.open('./wallet.dat', "main")
cursor = wallet.cursor()
rec = cursor.first()
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:4] == "key" and ord(key[0:1]) == 3:
        print "key 1st byte %d name %s Length %d" % (ord(key[0]) , key[1:4] , len(key))
        print "Public Key %s" % (key[4:].encode('hex'))
        hkey(value.encode('hex'))
        exit()
    rec = cursor.next()
```

Output

```
key 1st byte 3 name key Length 38
```

Public Key
 210200b36c9eb4535e21529b90d37385644fb419392d72b2442a8d42d11837b94f55
 Value d63081d30201010420d4a33a869d425eb5375deed0531fa1bb Length 494

The last key has a name called key. It is the most difficult key of all the keys. The first byte of the key is 3, the name is called key and the public key following has a length of 33 or 0x21 bytes.

Therefore, the total length of the key is 38 (1 + 3 + 1 + 33). The value has a length of 494 bytes, all gibberish for now.

The public key is not known and so is the value. All the same, it is the heart of the wallet.

Output

key 1st byte 3 name key Length 38
 Public Key 210200c9a60fc9733005bb37b20d49f9bc28a795cedac69d7f304d489405ff06cc5a
 Value d63081d3020101042068eb6830dc1a638c1cd3bdd387192f60 Length 494

It is easier to understand this key as its simpler, just a public key for now.

The Public Key in the Key Section of a Key called Key

```
ch2516.py
from bsddb3 import db
import subprocess
import json
lpubkey = []
lpubpool = []
def hkey(key):
    lpubkey.append(key[8:])
def hpool(value):
    lpubpool.append(value[24:])
wallet = db.DB()
#wallet.open('/Users/vijaymukhi/Desktop/wallet.dat', "main", db.DB_BTREE ,db.DB_RDONLY )
wallet.open('./wallet.dat', "main", db.DB_BTREE ,db.DB_RDONLY )
cursor = wallet.cursor()
rec = cursor.first()
rawtx = subprocess.check_output(["bitcoin-cli", "getaddressesbyaccount", ""])
lbaddr = json.loads(rawtx)
lpubkeyu = []
cnt = 0
cntpool = 0
for baddr in lbaddr:
    raw1 = subprocess.check_output(["bitcoin-cli", "validateaddress", baddr ])
    raw1 = json.loads(raw1)
    if len(raw1["pubkey"]) == 66:
        lpubkeyu.append("21" + raw1["pubkey"])
    else:
        print "Vijay Mukhi is a embecile first for"
while rec:
    key = rec[0]
```

```
value = rec[1]
if key[1:4] == "key" and ord(key[0:1]) == 3:
    hkey(key.encode('hex'))
    cnt = cnt + 1
elif key[1:5] == "pool":
    hpool(value.encode('hex'))
    cntpool = cntpool + 1
    pass
rec = cursor.next()
print "cntpool %d cnt %d" % (cntpool, cnt)
print "lpubkey Length %d lpublool Length %d lpubkeyu Length %d" % (len(lpubkey), len(lpubpool), len(lpubkeyu))
cnt1 = 0
for pub in lpubkey:
    if pub in lpubpool:
        print "(%d) %s in Pool Will be used later" % (cnt1, pub)
    else:
        if pub in lpubkeyu:
            print "(%d) %s Used by us in creating a Bitcoin Address" % (cnt1, pub)
        else:
            print "(%d) %s Vijay Mukhi is a embecile" % (cnt1, pub)
            cnt1 = cnt1 + 1
print cnt1
```

Output

```
cntpool 100 cnt 106
lpubkey Length 106 lpublool Length 100 lpubkeyu Length 6
(0) 210200b36c9eb4535e21529b90d37385644fb419392d72b2442a8d42d11837b94f55
in Pool Will be used later
(21) 21027f3fc87f0605e725272b757b0c86edb9833ffd8547ab67ac7f29d11ef5f3e9fb
Used by us in creating a Bitcoin Address
106
```

Running the program, ch2506.py once again shows that the keymeta key pairs have increased and there are 106 keys now.

In this program, we look out for the source, original location of the public key, which is assigned to the key named key. First, all the Bitcoin addresses in our wallet are obtained and placed in the lbaddr list. Then once again, the public keys used to create these Bitcoin addresses are put in the lpubkeyu list. The only modification here, is that whenever the first byte of the public key is 0x21 or 33 bytes or 66 characters, the missing length 21 is added at the start of the public key. This simplifies comparing two public keys.

In a while loop, every wallet key is scanned and checked for 2 keys, namely key and pool. Since there can be many key names starting with key, the length is also checked. The length must be 3 for key.

When a match is found, the hkey function is called. Here, the public key starting from the 8th position is added to the lpubkey list. In the end, the lpubkey list contains all the public keys found in the key called key.

When the key matches the word pool, the public key found at the 24th position in the value portion is added to the list lpubpool. Please note that it is not in the key section.

The lpubpool list now has all the public keys found in the pool key. The pool list lpubpool will always have 100 keys, the lpubkey list has 106 of them and it keeps growing. The smallest list is lpubkeyu, it has only 6 members.

The cnt variable gives a count on the number of keys named key present in our wallet. Similarly, the variable cntpool counts the number of keys called pool.

In a for loop, every public key present in the lpubkey list is accessed. The loop variable pub represents an individual public key. This loop will loop 106 times as we have that many public keys that were found in the key called key. There are three different types of if statement conditions or possibilities for the public keys in the list lpubkey.

The public key in the pub variable exists in the pool list. If it is, then it is confirmed that this public key has not been used to create a Bitcoin address, so far. That's because, when a pool public key is used, it gets discarded from the pool and a new public key is added at the end. The public key is available and can be used if it is in pool. This is the most frequent condition that will be true.

The next check is if the public key exists in the list, lpubkeyu. This list has the public keys which were used to generate Bitcoin addresses in the past. These public keys did not come from the moon but were present in the pool. The list is smaller in size, in our case its size is 6. If the address is present we display a message saying, "used by us". The public key displayed is in the list of public keys shown earlier.

Finally, if the last else is called, then there is an error as a public key in the key must be in the pool or would be used to generate a Bitcoin address. Any other value simply means an error in our understanding.

In passing, every public key in the key called name must be in the pool or used in the past to generate a Bitcoin address.

Given one value it is easy to determine the other value. Our pool list which is populated by the key called key will always have a minimum of 100 members. So far, we have created 6 Bitcoin addresses, so there will be 100 + 6 or 106 members in the list. The output has 106 public keys, 100 in the pool, 6 used by us and none speak the truth about Vijay Mukhi, which is no error called.

The new output is

Output

Pool cnt 100 key cnt 102

1

(10) 21024394d243b1358b8793ecff5c9d73de04b83b78d73392dca76f285fa72a5661e7 in Pool

(11) 210244e8644cb593375de665861ac95ed28867d242f4d3f701b30ec0643a596decbb3 Used by us
ch2517.py

Output

(24) 2102971174d2fd6fb688034b9824e75a051daaec753e4c85e4e66ae1d735590fd5f9 Vijay Mukhi is a embecile

There are 100 keys in the pool. The lpubkey has 102 members. There is only one member in the lpubkeyu list. The 102 Bitcoin addresses are scanned and checked against 100 + 1 or 101 Bitcoin addresses. Thus, one address in the lpubkey list will not be found in lpubpool or lpubkeyu.

How this public key made it into the list remains a mystery.

Understanding the Value of the Key called Key

```
ch2517.py
from bsddb3 import db
import hashlib
def chash(s):
    temp = hashlib.sha256(s).digest()
    final = hashlib.sha256(temp).digest()
    return final

wallet = db.DB()
wallet.open('./wallet.dat', "main")
cursor = wallet.cursor()
rec = cursor.first()
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:4] == "key" and ord(key[0:1]) == 3 :
        publickey = key[5:].encode('hex')
        print key[4:].encode('hex')
        print "%s:%d" % (value[0:5].encode('hex') , len(value))
        privatekey = value[1:215].encode('hex')
        hash = value[215:]
        final = publickey + privatekey
        ans = chash(final.decode('hex'))
        print ans == hash
        exit()
    rec = cursor.next()

wallet.close()
```

Output

```
210200b36c9eb4535e21529b90d37385644fb419392d72b2442a8d42d11837b94f55
d63081d302:247
True
```

Let's move closer to understanding what the value of the key named key represents. The first byte of the value is 0xd6 or 214 bytes. This is the length of the data following it. In our case, it is a serialized private key. Then there is a Sha256 hash value of 32 bytes. Therefore, the length of the value field is 247, $214 + 32 = 246$ plus one byte for the starting length of d6. It gives a total of 247 bytes.

The hash is a SHA256 double hash of the public key concatenated by the private key. The public key is part of the key name and not the value field. The public key starts with the length 0x21 which is removed and hence the extraction begins from the fifth and not the fourth byte onwards. The variable publickey stores the encoded public key starting from the fifth byte to the very end of the key.

While extracting the private key, the first length byte, 0xd6 is ignored. The private key length is always 0xd6 so it is hard coded in our code. The hash value starts 215 bytes from the start, another hard-coded value.

The private key is obtained by reading the bytes from 1 to 215 and then encoding them.

The public and private keys are concatenated and then decoded. Then the trusted chash function is used to give a double sha256 hash. The bytes from position 215 onwards to the end of the value field have the 32-byte hash, so these bytes are read and saved in variable hash. Thereafter, the hash which is the last 32 bytes of the value field is compared with the newly computed hash of the public and private key. They should be the same.

As an assignment, remove the exit function and check if the hashes match for all the keys called key. The most difficult key called key was made easier after reading the source and acknowledging the fact that the private key must be stored at someplace in the wallet.

Displaying the Value Portion of the Key called Key

```
ch2518.py
from bsddb3 import db
import bitcoin
import subprocess
import json
lpubkeyu = {}
lprivkey = {}
def hkey(key , value , lpubkeyu):
    publickey = key[5:].encode('hex')
    baddr = lpubkeyu.get(publickey, None)
    if baddr is None:
        return
    print "Key: Public Key is %s" % publickey
    print "Public Key Bitcoin address is %s" % baddr
    length1 = len(value)
    print "Length of value is %d" % length1
    fbyte = value[0:1].encode('hex')
    print "First Byte %s:%d" % (fbyte , int(fbyte,16))
    begin = value[1:9].encode('hex')
    print "Constant begin %s" % begin
    privatekey = value[9:41].encode('hex')
    print "private key %s" % privatekey
    middle = value[41:182].encode('hex')
    print "Constant middle %s" % middle
    publickey1 = value[182:215].encode('hex')
    print "Public Key %s" % publickey1
    hash = value[215:247].encode('hex')
    print "hash %s" % hash
    print "Public Keys are the same %s" % (publickey == publickey1)
    length = (len(fbyte) + len(begin) + len(privatekey) + len(middle) + len(publickey1) + len(hash))/2
    print "Our length is %d:%s" % (length , length1 == length)
    cprivatekey = privatekey + '01'
    print "Private Key Compressed (hex) is: " , cprivatekey
    wifprivatekey = bitcoin.encode_privkey(bitcoin.decode_privkey(cprivatekey, 'hex') , 'wif')
    print "WIF-Compressed is %s" % wifprivatekey
```

```
print "Found Private Key %s" % lprivkey.get(wifprivatekey , None)
print
print

rawtx = subprocess.check_output(["bitcoin-cli" , "getaddressesbyaccount" , ""])
lbaddr = json.loads(rawtx)

for baddr in lbaddr:
    raw1 = subprocess.check_output(["bitcoin-cli" , "validateaddress" , baddr ])
    raw1 = json.loads(raw1)
    if len(raw1["pubkey"]) == 66:
        lpubkeyu[raw1["pubkey"]] = baddr
    else:
        print "Vijay Mukhi is a embecile first for"
        raw2 = subprocess.check_output(["bitcoin-cli" , "dumpprivkey" , baddr ])
        lprivkey[raw2[:-1]] = baddr

wallet = db.DB()
wallet.open('./wallet.dat' , "main" )
cursor = wallet.cursor()
rec = cursor.first()
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:4] == "key" and ord(key[0:1]) == 3 :
        hkey(key, value , lpubkeyu)
        #exit()
    rec = cursor.next()
wallet.close()
```

Output

```
Key: Public Key is 027f3fc87f0605e725272b757b0c86edb9833ffd8547ab67ac7f29d11ef5f3e9fb
Public Key Bitcoin address is 1KGCWjm3ucJ1Vqfi7sKwYekuyJQjVVjSN
Length of value is 247
First Byte d6:214
Constant begin 3081d30201010420
private key 5e24a3dcf1a586649c5e18fe2abac8807db9464ca95dc68f98a6c6620f1ca941
Constant middle
a08185308182020101302c06072a8648ce3d0101022100ffffffffffffffffffffffffffffffffffff
ffff ffffffffefffffc2f300604010004010704210279be667ef9dcbbac55a06295ce870
b07029bfcdb2dce28d959f2815b16f81798022100fffffffffffffffffffffffffffffebaedce6
af48a03bbfd25e8cd0364141020101a124032200
Public Key
027f3fc87f0605e725272b757b0c86edb9833ffd8547ab67ac7f29d11ef5f3e9fb
hash cc2b5542ad087e80ef3cb3bf69a38913887781fa1587b240b0aa44d902a597a0
Public Keys are the same True
Our length is 247:True
Private Key Compressed (hex) is: 5e24a3dcf1a586649c5e18fe2abac8807db9464ca95dc68f98a6c6620f1ca94101
```

WIF-Compressed is KzNiJAPh47MjbpQ1EbgnisaiSHJA8EwzDzwGp44MGyChZ9q35LkZ
 Bitcoin address is 1KGCWjm3ucJ1Vqfi7sKwYekuyJQjVVjSN

Total cnt 106

This program creates two empty dictionaries, `lpubkeyu` and `lprivkey` to substantiate our explanations. As before, the `lbaddr` list has Bitcoin addresses created by the wallet. There is a loop which accesses each of these addresses. In one of the earlier programs, we created a list of public keys using the `bitcoin-cli` client.

Now let's do things a little differently. First, a key is added to the `lpubkeyu` dictionary where the key is a public key and the value is a Bitcoin address. The `else` clause for error check never ever gets called because the length of the public key returned is always 66.

The second dictionary called `lprivkey` is also filled up. Here the key is a private key and the value is the same Bitcoin address, `baddr`. The idea is that given a public key or a private key, one can obtain the Bitcoin address that was responsible for creating it in the first place. The last pesky enter byte is removed.

For the last time in this chapter, the same while cursor loop is executed. The same `hkey` function is called with 3 parameters, the key, value and the public key dictionary `lpubkeyu`. The dictionary of private keys, `lprivkey`, is not given to the function, out of protest over Python's methods on handling global variables.

The function `hkey` has code that explains the 247 bytes of the value field. The public key is the fifth byte from the start of the key called `key`, this public key is pulled out and displayed. The first 4 bytes of the key are taken up by the key named `key` and there is only one public key following this key name. The public key also does not start with the length byte, 0x21 but with 02.

The `get` function of the dictionary `lpubkeyu` checks if the public key has been used earlier to create a Bitcoin address. If yes, the Bitcoin address is accessed in variable `baddr`. If no, then `None` is returned.

Our interest lies in those keys called `key` which are used for creating Bitcoin addresses only. If the key has not been used, then there is a return. Comment out this return and watch the result. There are 6 outputs due to 6 Bitcoin addresses we have created in the past. The function `hkey` gets called 106 times and the return statement gets executed 100 times. The displayed Bitcoin address starting with 1KG is the first address in the list. Verify that the public key in the key portion also matches with the output of the command `validateaddress`.

The public key from the key field and the Bitcoin address created by it are displayed.

The value field is reconfirmed to be 247 bytes large. The length of encoded values is generally half of the actual decoded size.

The first byte of the value field is always a hard-coded value of 214 bytes or 0xd6. This indicates that the next 214 bytes are to be treated as one entity. This number is added more because the code written in C++ needs these bytes. The next 8 bytes are read into a variable called `begin`. This format is again a contribution from the source code. Google or books don't help here.

The European world created an encoding scheme called BER or Basic Encoding Rules. This encoding scheme was very difficult to use, hence a scaled down version called DER or Distinguished Encoding Rules was created. The private key is stored using DER and the value in `begin` are the initial bytes of a DER encoded data object. This is done before.

We would have loved to explain DER encoding to you since we have tutorial on it, but the Bitcoin source uses hardcoded values. To add insult to injury, the Bitcoin developers use the phrase "are very ugly" in the source code.

The hardcoded ugly values used in the Bitcoin code are called constants. These initial bytes take up 8 bytes. They start with a 0x30, which is a known devil in the DER world.

Then comes the 32-byte private key which is stored in the variable or better still parameter called `value`. This private key

starts from byte 9 and ends at byte 41, for a total of 32 bytes. It is displayed in its encoded form. Then, another 141 bytes of a constant are read; the Bitcoin source calls it middle. These are hardcoded values.

The next 33 bytes belong to the public key. These bytes are from position, 182 to 215. This public key matches the one extracted from the key. The variable `publickey1` stores this public key. The values in variables `public key` and `publickey1` are checked and they match. The last 32 bytes of the value field is the hash value of the public key and private key.

The length of the individual fields is checked to add up to 247. Then the length of the variables `fbyte`, `begin`, `privatekey`, `middle`, `pubkey1` and `hash` are added. The encoded length is divided by 2 to give a decoded length. The last comparison confirms that the newly calculated length in variable `length1` matches the value in variable `length`, or both are 247 bytes.

Next check is on the private key. Do we have the same one? We know that the private key creates a public key from which the `RipeMD` hash value is obtained and then the base58 encoded Bitcoin address.

```
bitcoin-cli dumpprivkey 1KGCWjm3ucJ1Vqfi7sKwYekuyJQjVVjSN
KzNiJAPh47MjbpQ1EbgnisaiSHJA8EwzDzwGp44MGyChZ9q35LkZ
```

To the private key given by the above command, we first add a 01 to the end. 01 at the end denotes a compressed private key.

Following our learning experience of coding from the internet, two functions are called in one. The function `decodeprivkey` is used to do what its name suggests, decode the private key to hex. Then the same private key is encoded to the wif format.

A sigh of relief as this recently calculated wif private key is present in the dictionary of private keys, `lprivkey`; the one which was populated at the start of our code.

The Bitcoin address matches the one obtained from the public key.

Only one output is given due to space restrictions. If not clear, please see this output:

Output

```
Key: Public Key is 0244e8644cb593375de665861ac95ed28867d242f4d3f701b30ec0643a596dec3
Public Key Bitcoin address is 192VaoqRQhYyJSp7hXYWfjpKtvG42XQYW8
Length of value is 247
First Byte d6:214
Constant begin 3081d30201010420
private key 7272948769101d41f7cf5fe36209114bc1489251ff244a22d1f453881e99ea0e
Constant middle
a08185308182020101302c06072a8648ce3d0101022100ffffffffffffffffffffffffffff
ffffffffffffffffffffc2f300604010004010704210279be667ef9dcbbac55a06295ce870
b07029bfcdb2dce28d959f2815b16f81798022100fffffffffffffffffffffffffffffebaedce6
af48a03bbfd25e8cd0364141020101a124032200
Public Key 0244e8644cb593375de665861ac95ed28867d242f4d3f701b30ec0643a596dec3
hash 928062399b201fb382f473f815ccf73a07778ebf98a0164358908097055b5010
Public Keys are the same True
Our length is 247:True
Private Key Compressed (hex) is:
7272948769101d41f7cf5fe36209114bc1489251ff244a22d1f453881e99ea0e01
WIF-Compressed is
L14BWZo6b3du2eLPxVMLUT7bcApH2WpGrAJLrtmQQwMXQyBaDatS
Bitcoin address is 192VaoqRQhYyJSp7hXYWfjpKtvG42XQYW8
```

The Bitcoin address displayed is ours, 192V.

Accounting for all the Keys in Our Wallet

```

ch2519.py
from bsddb3 import db
import subprocess
wallet = db.DB()
wallet.open('./wallet.dat', "main")
cursor = wallet.cursor()
rec = cursor.first()
cnt = 0
cnttx = 0
cntdefaultkey = 0
cntkeymeta = 0
cntpool = 0
cntname = 0
cntpurpose = 0
cntversion = 0
cntacc = 0
cntbestblock = 0
cntorderposnext = 0
cntbestblock_nomerkle = 0
cntkey = 0
cntminversion = 0
while rec:
    key = rec[0]
    value = rec[1]
    if key[1:3] == "tx":
        cnttx = cnttx + 1
    elif key[1:11] == "defaultkey":
        cntdefaultkey = cntdefaultkey + 1
    elif key[1:4] == "key" and ord(key[0:1]) == 3:
        cntkey = cntkey + 1
    elif key[1:8] == "keymeta":
        cntkeymeta = cntkeymeta + 1
    elif key[1:5] == "pool":
        cntpool = cntpool + 1
    elif key[1:5] == "name":
        cntname = cntname + 1
    elif key[1:8] == "purpose":
        cntpurpose = cntpurpose + 1
    elif key[1:8] == "version":
        cntversion = cntversion + 1
    elif key[1:4] == "acc":
        cntacc = cntacc + 1
    elif key[1:10] == "bestblock" and ord(key[0]) == 9:

```

```
cntbestblock = cntbestblock + 1
elif key[1:13] == "orderposnext":
    cntorderposnext = cntorderposnext + 1
elif key[1:19] == "bestblock_nomerkle":
    cntbestblock_nomerkle = cntbestblock_nomerkle + 1
elif key[1:11] == "minversion":
    cntminversion = cntminversion + 1
else:
    print "Error not handled key"
    print key
    print key.encode('hex')
    print value.encode('hex')
    cnt = cnt + 1
    rec = cursor.next()
```

```
tot = cnttx + cntdefaultkey + cntkeymeta + cntpool + cntname + cntpurpose + cntversion +
cntacc + cntbestblock + cntorderposnext + cntbestblock_nomerkle + cntkey + cntminversion
print "Total Number of keys in wallet are %d" % cnt
print "Key tx %d" % cnttx
print "Key defaultkey %d" % cntdefaultkey
print "Key keymeta %d" % cntkeymeta
print "Key pool %d" % cntpool
print "Key name %d" % cntname
print "Key purpose %d" % cntpurpose
print "Key version %d" % cntversion
print "Key acc %d" % cntacc
print "Key bestblock %d" % cntbestblock
print "Key orderposnext %d" % cntorderposnext
print "Key cntbestblock_nomerkle %d" % cntbestblock_nomerkle
print "Key key %d" % cntkey
print "Key minversion %d" % cntminversion
print "Total Number of keys %d %s" % (tot, cnt == tot)
```

Output

```
Total Number of keys in wallet are 611
Key tx 280
Key defaultkey 1
Key keymeta 106
Key pool 100
Key name 6
Key purpose 6
Key version 1
Key acc 1
Key bestblock 1
Key orderposnext 1
Key cntbestblock_nomerkle 1
Key key 106
```

```
Key minversion 1
Total Number of keys 611 True
```

We have accounted for all the 611 keys in our wallet. What's missing here is using a password to encrypt the wallet and having one large program to display the entire wallet.

We ran the same code on a wallet created with Bitcoin Core Version 0.15. The bitcoind server shows us the following. This wallet is empty.

```
2017-10-13 17:51:49 Performing wallet upgrade to 60000
2017-10-13 17:51:53 keypool added 2000 keys (1000 internal), size=2000 (1000 internal)
```

This spells trouble as we have 2000 keys in our pool, not a measly 100. When we ran the earlier program, the output received is given below.

```
Error not handled key
hdchain
076864636861696e
02000000e80300002b0ff4e2e4525cde9eda632c210f256e302f3796e8030000
Total Number of keys in wallet are 6007
Key tx 0
Key defaultkey 0
Key keymeta 2001
Key pool 2000
Key name 0
Key purpose 0
Key version 1
Key acc 0
Key bestblock 1
Key orderposnext 0
Key cntbestblock_nomerkle 1
Key key 2001
Key minversion 1
Total Number of keys 6006 False
```

Only one extra key called hdchain gets created. We have no idea what the 32-byte long value of this key represents. The source code talks of this being two ints and a 160-bit hash.

CHAPTER 26

Rev/Undo files

With this chapter, we end our explanations of the all the major components created in the Bitcoin ecosystem.

For every blk.dat file in the blocks folder, there is a corresponding rev.dat file. Copy the first undo or rev file rev00000.dat from the blocks folder into your current folder.

Reading rev00000.dat File from Disk

```
ch2601.py
from cfuncs import *
f = open("./rev00000.dat" , "rb")
for i in range(0,3):
    magic = rint(f)
    print "Magic Number %x" % magic
    sizer = rint(f)
    print "Size of Undo Block %d" % sizer
    undoblock = rbytes(f , sizer)
    print "Undo data %s" % undoblock.encode('hex')
    revhasho = rbytes(f , 32)
    print "Hash on disk %s" % revhasho.encode('hex')
    print
```

Output

```
Magic Number d9b4bef9
Size of Undo Block 1
Undo data 00
Hash on disk 90c1d8d0ad21e0fc9266492be7fdb3d07348a74aa6f1655d478589178a3589ad

Magic Number d9b4bef9
Size of Undo Block 1
Undo data 00
Hash on disk f2c2727e535da5d314238c5e841b4346af303cecc8356bf2b739c8cd786b34a3

Magic Number d9b4bef9
Size of Undo Block 1
Undo data 00
Hash on disk cf0f972a59caa6b1893b9db2c2598d86e12634a1b74617aa063ba07bcb734468
```

This program explains the file format of an undo file is. Open the file rev00000.dat as always, in your favorite hex editor. The first 4 bytes are the same magic number, 0xd9b4bef9 as seen in every blk.dat files. Then comes a 4-byte number

for the size of the undo bytes. Fortunately, all integers are stored using 4 bytes of disk space and not a variable number of bytes.

In our case, all the initial blocks have a size field of value 1. The undo byte following has a value of 0 which means that the undo buffer bytes are absent. These bytes are the heart and soul of this chapter but will be explained a little later in the chapter.

This format ends with a hash which is 32 bytes large. A very simple file format. All the hashes shown are different.

Validating the Hash Value Found in the Rev File

```
ch2602.py
from cfuncs import *
import subprocess
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
f = open("./rev00000.dat", "rb")
for i in range(0,3):
    magic = rint(f)
    sizer = rint(f)
    undoblock = rbytes(f, sizer)
    revhasho = rbytes(f, 32)
    print "Hash on disk is %s" % revhasho.encode('hex')
    hash = subprocess.check_output(["bitcoin-cli", "getblockhash", "%s" % i])
    hash = hash[:-1]
    hash = reversehash(hash)
    hashstr = hash + undoblock.encode('hex')
    revhash = chash(hashstr.decode('hex'))
    print "Hash calculated %s" % revhash.encode('hex')
    if revhash == revhasho:
        print "Hashes Match for block no %d" % i
    else:
        print "Hashes do not match"
```

Output

```
Hash on disk is
90c1d8d0ad21e0fc9266492be7fdb3d07348a74aa6f1655d478589178a3589ad
Hash calculated
90c1d8d0ad21e0fc9266492be7fdb3d07348a74aa6f1655d478589178a3589ad
Hashes Match for block no 0
Hash on disk is
f2c2727e535da5d314238c5e841b4346af303cecc8356bf2b739c8cd786b34a3
Hash calculated
f2c2727e535da5d314238c5e841b4346af303cecc8356bf2b739c8cd786b34a3
Hashes Match for block no 1
Hash on disk is
cf0f972a59caa6b1893b9db2c2598d86e12634a1b74617aa063ba07bcb734468
Hash calculated
```

cf0f972a59caa6b1893b9db2c2598d86e12634a1b74617aa063ba07bcb734468
Hashes Match for block no 2

This program looks at the calculation of the last 32-byte hash. As before, the 32-byte hash found in the rev file is first displayed. The variable undoblock contains the undo bytes and the variable revhasho, the 32-byte hash.

Then, our good old friend bitcoin-cli with the getblockhash command is used to return the hash of the block. The getblockhash command takes a string and not a number and hence the %s modifier is used to convert a number to a string.

The loop variable i stands for the block number. The last byte of the returned string has an enter which is removed by the slice operator: -1. This is seen earlier.

This is our favorite bugbear, we have not yet been able to figure out how to use the returned hash value, take it as is or reverse it. If there is some pattern, we are yet to understand it, coz we have still not understood the principles behind reversing the hash. The variable revhasho stores the hash found in the rev file.

Now, a string is required which starts with the hash value of the block and then the undo block bytes. This order is most important here. As the undo bytes are in the decoded form, they are to be encoded first. The concatenated outcome to the two strings is saved in the variable, hashstr.

Thereafter, the Sha256 hash value is computed using the same old chash function. As this returned hash is a decoded string or a string with actual bytes, the encoded string is displayed and they match to the T. The if statement checks if they are the same. In our case, the first three blocks match. This confirms that the hash or checksum are stored on disk to check if any changes have been made to the undo bytes.

The Bitcoin core validates these undo bytes and then adds them to the block hash at the end. The Sha256 double hash is then stored on disk.

Now comes the difficult part, verifying each and every rev hash stored on disk. There are about a half a million hashes to verify.

First and foremost, the program bitcoin-cli cannot be used to obtain the block hashes of half a million blocks. It is simply not practical as it will be very slow. It is not possible to scan every block on disk and then to compute the hash. The hash also would be inaccurate as the blocks contain all sorts of orphan blocks and more. So, a table called blockhash is created with the block number and the block hash.

As a one-time exercise, having a table with an index on the block number is more feasible as the hash of any block is easily accessible plus searches are much faster.

The rev files are also a cousin of the blk files and therefore it is naïve to assume that the blocks in the rev file will ever be in order. So, we cannot use them at all as they will contain out of order blocks. The blocks supplied by the miner is out of our control so they can be unorganized, but these rev files are written by code running on our computer. It is expected of these blocks to be stored in some order. Nevertheless, no one in the Bitcoin code, is reading these rev files, as they are considered insignificant. They get created when the Bitcoin blocks are download on the disk. These rev files are rewritten only when the blocks are re-indexed.

The Bitcoin core reads these rev files when the blocks at a certain level are to be verified. Now, the verification simply checks for a hash match. There is one more case called DisconnectBlock which will be tackled later.

So, to summarize, the program Bitcoin-cli cannot be used and the blocks in a rev file have no ordering and therefore they cannot be read sequentially.

To verify the hash, the hash value of a block number is required. The rev file blocks like those in the blk files have no record numbers embedded in them.

Also, the Bitcoin core does not read the blk and rev files sequentially. The information stored of every block in the index folder is its hash, height or record number, the file number in which it is physically present and the position of its starting point in the rev and blk files. So, the next best option is to read the index leveldb database and figure out the position in the rev file where this block starts and then concatenate the undo data and the block hash.

Hey! We are getting ahead of ourselves!

The chainstate folder having the UTXO set has all the unspent outputs. So, what about the spent outputs? it is not practical to scan 80 GB of block data to determine the spent outputs.

The genesis block has no undo data because it cannot undo anything. The genesis block is created by code only and it is not spendable. One more reason, the rev file shows the first block as the start and not the 0th or the genesis block.

To get clarity on these issues, the source code comes handy. It clearly states that to compute a hash value, the block hash value of the previous block must be used and not the current block. The hash of the current block determines the position or the location in the rev file for reading the undo data of the blocks. It is then hashed with the hash of the previous block.

So, use the current block hash value to determine where in the rev file the data of the undo block resides, but use the previous blocks hash to compute the hash. Said it again!

```
create table blockhash (blkno integer , hash varchar(64));
```

Storing the Block Number and Hash of Every Block into a SQL Table

```
ch2603.py
import subprocess
import psycopg2

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

blkno = 0
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
for blkno in range(0, 500000):
    blknos = "%d" % blkno
    try:
        hash = subprocess.check_output(["bitcoin-cli", "getblockhash", blknos])
    except:
        print "Quitting out"
        conn.commit()
        exit()
    hash = hash[:-1]
    hash1 = reversehash(hash)
    s1 = "insert into blockhash values(%d , '%s' )" % (blkno , hash)
    cur.execute(s1)
    blkno = blkno + 1
    if blkno % 10000 == 0:
        print blkno
```

The blkno variable used in the for statement represents the block hash number. Using this variable and the getblockhash

command, one block hash is acquired. Subsequently, the block number and the hash are stored in a table called blockhash.

This process eliminates any further use of the bitcoin-cli program.

The commit command is not required, this happens automatically when we gracefully quit out. But it is good programming to commit the data in a database once, just before quitting.

```
create index zz on blockhash(blkno);
```

The above command creates an index on the field blkno in the table blockhash. This makes the mare go faster.

No Valid Undo Block Data for the Genesis Block

```
ch2604.py
import leveldb
import time
import struct
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)
def index(ans):
    (nVersion , offset) = base128(ans , 0)
    print "nVersion%d" % (nVersion)
    (nHeight, offset) = base128(ans , offset)
    print "nHeight %d" % (nHeight)
    (nStatus, offset) = base128(ans, offset)
    print "nStatus %x:%d" % (nStatus , nStatus)
    (nTx, offset) = base128(ans, offset)
    print "nTx %d" % (nTx)
    (nFile, offset) = base128(ans, offset)
    print "nFile %d" % (nFile)
    (nDataPos, offset) = base128(ans, offset)
    print "nDataPos %d" % (nDataPos)
    (nUndoPos, offset) = base128(ans, offset)
    print "nUndoPos %d" % (nUndoPos)
    print "Length of offset %d:%d" % (len(ans) - offset, len(ans))
    (version,prevblockhash , merklehash , creationtime , difficulty , nonce) =
    struct.unpack("I32s32sIII" , ans[offset : offset + 80 ])
    print "Version %d" % version
    print "PrevBlock Hash %s" % prevblockhash[:: -1].encode('hex')
    print "Merkle Root %s" % merklehash[:: -1].encode('hex')
```

```

    print time.ctime(creationtime)
    print "Difficulty%d" % difficulty
    print "Nonce%d" % nonce
    final = offset + 80
    print "Final byte %s" % final

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/index" )
hash = '00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048'
hash = reversehash(hash)
hash = '62' + hash
hash = hash.decode('hex')
print db.Get(hash).encode('hex')
index(db.Get(hash))
hash = '000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f'
hash = reversehash(hash)
hash = '62' + hash
hash = hash.decode('hex')
index(db.Get(hash))

```

Output

```

nVersion 110200
nHeight 1
nStatus 1d:29
nTx 1
nFile 0
nDataPos 301
nUndoPos 8
Length of offset 80:90
Version 1
PrevBlock Hash 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
Merkle Root 0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098
Fri Jan 9 08:24:25 2009
Difficulty 486604799
Nonce 2573394689
Final byte 90
nVersion 110200
nHeight 0
nStatus b:11
nTx 1
nFile 0
nDataPos 8
nUndoPos 1
Length of offset 79:88
Traceback (most recent call last):
  File "dummy.py", line 54, in <module>

```

```
bitcoin-cli getblockhash 0
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

```
bitcoin-cli getblockhash 1
00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048
```

The getblockhash command retrieves the block hashes for block 0 and 1, but we are well versed with these block hashes so we hard-code them. You can check the hashes using any blockchain explorer. The hash is reversed and a 'b' or 0x62 is added at the start of the hash to make it a valid block key for the index folder.

Then the hash value is decoded. The Get function retrieves all the block hash data stored in the index leveldb database. The index function deciphers this value. The last 80 bytes and the 8 bytes from the start of the block have the same block structure.

For block number 1, all is fine. But for the genesis block or block number 0, the block structure is 79 bytes in size and not 80 bytes. Therefore, an exception is thrown. One more reason why block 0 is not present in the rev file. Also, the start of the undo data is at position 1, which does not make any sense for now. It is for the genesis block.

The variable nDataPos has a valid value of 8, but the variable nUndoPos has a value of 1. The undo position for block number 1 however, has a value of 8 and not 0 or 1. The reason being, the first 8 bytes are used by the magic number and size of the undo bytes. For the first block, the undo data starts at byte 9 or byte 8, if counting starts from 0.

The index folder also gives the rev and block file number in the nFile variable along with the start position of the rev undo data in this file. A small correction, not where the block starts, but where the undo data starts, an offset of 8.

Obviously, the undo block starts -8 bytes from the same location as the undo data. Since the undo data does not point to the size of the undo data field, there is a back shift. The size of the undo block data is gathered from here and it keeps changing. Had the position in the index key been the start of the block, it would be simpler obtaining the size of the undo data.

Scanning all the Undo Blocks We Have

```
ch2605.py
import leveldb
import psycopg2
from cfuncs import *

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)
def index(ans):
    (nVersion , offset) = base128(ans , 0)
```

```

(nHeight, offset) = base128(ans , offset)
(nStatus, offset) = base128(ans, offset)
(nTx, offset) = base128(ans, offset)
(nFile, offset) = base128(ans, offset)
(nDataPos, offset) = base128(ans, offset)
(nUndoPos, offset) = base128(ans, offset)
return (nFile , nUndoPos)

conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index')

for blkno in range ( 1 , 500000):
    #print "Block Number is %d" % blkno
    s1 = "Select hash from blockhash where blkno = %d" % (blkno )
    cur.execute(s1)
    row = cur.fetchone()
    hash = row[0]
    hash = reversehash(hash)
    #print "Block Hash for block %d %s" % (blkno , hash)
    s2 = "Select hash from blockhash where blkno = %d" % (blkno - 1 )
    cur.execute(s2)
    row1 = cur.fetchone()
    hash1 = row1[0]
    hash1 = reversehash(hash1)
    #print "Block Hash for block %d %s" % (blkno - 1 , hash1)
    hashb = "62" + hash
    hashb = hashb.decode('hex')
    ans = db.Get(hashb)
    (nFile , nUndoPos) = index(ans)
    #print "undoPos %d" % nUndoPos
    fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/rev%05d.dat" % nFile
    #print fname
    fr = open(fname , "rb")
    try:
        fr.seek(nUndoPos - 8 , 0)
    except:
        print "Its all over"
        exit()
    magic = rint(fr)
    if magic == 0:
        print "Major Error bailing out"
        exit()
    sizer = rint(fr)
    undoblock = rbytes(fr , sizer)
    hashstr = hash1 + undoblock.encode('hex')
    revhash = chash(hashstr.decode('hex'))
    revhasho = rbytes(fr,32)

```

```
if blkno % 10000 == 0:
    print "Block Number %d" % blkno
    if revhash == revhasho:
        #print "(%d) All is good" % blkno
        pass
    else:
        print "Calculated checksum %s" % revhash.encode('hex')
        print "Rev file checksum %s" % revhasho.encode('hex')
```

Output

```
Block Number is 1
Block Hash for block 1
4860eb18bf1b1620e37e9490fc8a427514416fd75159ab86688e9a8300000000
Block Hash for block 0
6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000
undoPos 8
/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/rev00000.dat
(1) All is good
Block Number is 2
Block Hash for block 2
bdd99ccfda39da1b108ce1a5d70038d0a967bacb68b6b63065f626a00000000
Block Hash for block 1
4860eb18bf1b1620e37e9490fc8a427514416fd75159ab86688e9a8300000000
undoPos 49
/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/rev00000.dat
(2) All is good
```

This program puts everything together. The blkno variable starts with a value of 1, and not 0, then gets a value of 2 and stops at a large number randomly chosen. The first step taken is to retrieve the hash of the block number 1 from the blockhash table into the hash variable. Then the block hash of the block 0 or the previous block is retrieved and placed into the hash1 variable. The value in the blockno variable is reduced by 1 to retrieve this previous block hash.

The hash value of the current block is in variable hash and that of the previous block is in variable hash1. These hashes must be reversed.

A character 'b' is prefixed to the block hash only. The reason being, the undo data of the current block is required and not the previous block.

The hash is decoded and then the Get function is used to return the value field. The return value is stored in a variable called ans. Once again, it is the undo block data of the current block only. The index function is called but it returns two values only. These values are the file number and the position in the rev file, where this block's undo data starts. Everything else is ignored.

The nFile variable has the handle to the rev file that contains the undo data of the block. Then the position in the file identified by the undo position from the index function is located. Though the undo data start position is known, its size or length is still a mystery.

There is a jump backwards of 8 bytes and not 4 bytes. The extra jump is to read the magic number, an error check. The size of the undo data stored in the next 4 bytes is then read and placed in the undoblock variable. Two strings are concatenated, namely the string hash1 which is the block hash for the block 0 or the previous block number and the undo block data in the undoblock variable.

The chash function creates the hash string and computes the hash, this value is assigned to the revhash variable. This value must match the hash value stored on disk. The variables revhasho and revhash must be the same. There is no error output so it has passed. Uncomment the print statements to see some more data.

There are many blocks in the rev files, but we will never get a count. One way out is to simply reuse the earlier hash. Outside the for loop, we compute the hash of block 0 once. Then the hash of block 1 is found and stored it in this hash1 variable. This value is used to read the index folder and the hash of block 0 is used to compute the hash value. At the end of the loop, the variables hash1 value is set to the variables hash value. Pardon the pun on hashes. This saves one SQL read command to the database.

If we must choose between complicated code and inefficient code that is easier to understand, we will opt for code that is simpler to understand even though it is inefficient.

Taking our criticism one step further, one can read all the block key value pairs first or use the try catch to catch exceptions. Then quit out gracefully. A piece of advice, never use large random numbers to end a loop.

```
select count (*) from blockhash;
count
-----
489644
(1 row)
```

There are nearly half a million blocks!! Soon, our code will not take the latest blocks into account. Change it to a million to future proof everything.

```
create table blockhash (blkno integer , hash varchar(64) , notran integer);
create index zz1 on blockhash(blkno);
```

Adding the Number of Transactions Per Block into the Blockhash Table

```
ch2606.py
import subprocess
import psycpg2
import json
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

blkno = 0
conn = psycpg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
for blkno in range(0, 500000):
    blknos = "%d" % blkno
    try:
        hash = subprocess.check_output(["bitcoin-cli" , "getblockhash" , blknos])
    except:
        print "Quitting out"
        conn.commit()
        exit()
    hash = hash[:-1]
    block = subprocess.check_output(["bitcoin-cli" , "getblock" , hash])
```

```
block = json.loads(block)
notrans = len(block['tx'])
s1 = "insert into blockhash values(%d , '%s' , %d )" % (blkno , hash , notrans)
cur.execute(s1)
blkno = blkno + 1
if blkno % 10000 == 0:
    print blkno
    conn.commit()
```

Output

440000

450000

A table is created with the same name, blockhash after dropping the earlier blockhash table. There is an addition of an integer field called notran. This field will hold the number of transactions in a block.

Every block must have at the very least one transaction, the Coinbase transaction. This transaction type is very special, here the input section does not matter and the output section contains at least one output, the mining reward.

Given a block hash, the command getblock returns the entire block data, including a list called tx that contains the transaction hashes of a block. This is only the list of transactions, not the data. The len function returns the length of the list or the number of transactions in that block.

This program takes much longer than the last one because an external program, bitcoin-cli is called, not once but twice. Running time measured in hours.

A Block with only a Coinbase Transaction must not have Undo Block Data

```
ch2607.py
import leveldb
import pycpg2
from cfuncs import *

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def index(ans):
    (nVersion , offset) = base128(ans , 0)
    (nHeight, offset) = base128(ans , offset)
    (nStatus, offset) = base128(ans, offset)
    (nTx, offset) = base128(ans, offset)
```

```

(nFile, offset) = base128(ans, offset)
(nDataPos, offset) = base128(ans, offset)
(nUndoPos, offset) = base128(ans, offset)
return (nFile , nUndoPos)

conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index')
notran = 0
nosizer = 0
nonormal = 0
nobad = 0
for blkno in range ( 1 , 500000):
    s1 = "Select hash, notran from blockhash where blkno = %d" % (blkno )
    cur.execute(s1)
    row = cur.fetchone()
    hash = row[0]
    hash = reverseahash(hash)
    hashb = "62" + hash
    hashb = hashb.decode('hex')
    try:
        ans = db.Get(hashb)
    except:
        break
    notran = row[1]
    if notran >= 2:
        notran = notran + 1
    (nFile , nUndoPos) = index(ans)
    fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/rev%05d.dat" % nFile
    fr = open(fname , "rb")
    try:
        fr.seek(nUndoPos - 8 , 0)
    except:
        break
    magic = rint(fr)
    if magic == 0:
        print "Major Error bailing out"
        break
    sizer = rint(fr)
    if sizer > 1 and notran == 1:
        print "Houston we have a problem at block number %d sizer %d" % (blkno , sizer)
    elif sizer > 1 and notran >= 2:
        nonormal = nonormal + 1
    else:
        nobad = nobad + 1
    if sizer == 1:
        nosizer = nosizer + 1

```

```
if blkno % 10000 == 0:
    print "Block Number %d" % blkno

    print "Block Number %d nosizer %d nonormal %d nobad %d %d" % (blkno, nosizer , nonormal ,
        nobad , nonormal + nobad)
```

Output

Block Number 450000

Block Number 453866 nosizer 87635 nonormal 366230 nobad 87635 453865

This program is mostly a copy of a program from an earlier chapter. It has a loop which reads the hash and number of transactions from the blockhash table. The loop goes on for about a half a million times.

As before, the hash value is reversed and a small letter 'b' is added at the beginning of the block hash. Then, it is decoded and used as a key to the index leveldb database. The value obtained after calling the index function gives access to a certain rev file this block resides in. The file pointer must be placed exactly where the undo data of the block is stored.

After moving back 8 bytes from where the index folder returns a position, the magic number is checked and most importantly the size of the undo block is retrieved in variable `sizer`. It must be noted that if there is no undo block data, the size of the data is 1 and not 0. The reason being that an empty undo block has a size of 1 and a data byte has a 0.

The next task is to check if the undo block has some data other than 0. If yes, then the size of the undo block is 2 bytes or more. Another check is, if the number of transactions in that block is 1. This could only mean that the block contains only a Coinbase transaction and nothing else.

To our surprise, there was not a single case where a block had a Coinbase transaction only and some undo block data associated with it. This print statement never gets displayed thus confirming the above.

A Coinbase transaction simply cannot be undone. A block with a Coinbase transaction will never have any undo data associated with it.

The program processes 489616 blocks. There is a count kept on the blocks that have undo data in them as then the `sizer` variable will be larger than 2. Also, obviously, these blocks have a transaction count larger than 2. Most of our blocks have undo data associated with them. Hence, the `nonormal` variable has a value of 401623 blocks.

There is a count kept on the undo blocks that have no undo data as well and these form a small percentage. This value is stored in the variable `nobad` which is part of the `else` statement. Most of the initial blocks have no undo data but are still present in the rev file. These add up to 87992 blocks.

The sum of the both the variables `nonormal` and `nobad` will add up to the number of blocks processed, 489615. The earlier outputs shows you older data. This is what we see now.

Block Number 489616 nosizer 87992 nonormal 401623 nobad 87992 489615

If all the Bitcoin blocks had a Coinbase transaction, then the rev files would carry no undo data.

So far, the inputs were treated as secondary citizens compared to the outputs. But it's time to undo the wrong. An input is more important than an output because a) it points to an output b) it also carries a signature to validate the transfer and c) it brings in money. For the time being, the outputs in the rev/undo files will be ignored. But outputs come first.

Let's understand the `try except`.

The program may throw an exception. The reason for this blockhash key error is because the blockhash key is not present in the index folder. The index folder in our case is not new and has old data. The blocks in the blockhash table

are more than the blockhash keys present in the index folder. To update the index folder, shut down bitcoind and recopy the index folder. Normally, it should not happen. Too lazy !!!!

The Undo Data Starts with the Number of Transactions in the Block - 1

```

ch2608.py
import leveldb
import psycopg2
from cfuncs import *

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def index(ans):
    (nVersion , offset) = base128(ans , 0)
    (nHeight, offset) = base128(ans , offset)
    (nStatus, offset) = base128(ans, offset)
    (nTx, offset) = base128(ans, offset)
    (nFile, offset) = base128(ans, offset)
    (nDataPos, offset) = base128(ans, offset)
    (nUndoPos, offset) = base128(ans, offset)
    return (nFile , nUndoPos)

conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index')
for blkno in range ( 1 , 500000):
    s1 = "Select hash, notran from blockhash where blkno = %d" % (blkno )
    cur.execute(s1)
    row = cur.fetchone()
    hash = row[0]
    notran = row[1]
    hash = reversehash(hash)
    hashb = "62" + hash
    hashb = hashb.decode('hex')
    try:
        ans = db.Get(hashb)
    except:
        break

```

```
(nFile , nUndoPos) = index(ans)
fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/rev%05d.dat" % nFile
fr = open(fname , "rb")
fr.seek(nUndoPos - 8 , 0)
magic = rint(fr)
sizer = rint(fr)
undoblock = rbytes(fr , sizer)
if sizer >= 2:
    first = ord(undoblock[0:1])
    if first == 253:
        second = ord(undoblock[1:2])
        third = ord(undoblock[2:3])
        #print "%d %d %d" % (second , third , second * 1 + third * 256)
        first = second * 1 + third * 256
    if first != notran - 1 :
        print "Houston problem at block number %d first %d notran %d " % (blkno , first , notran)
    if blkno % 10000 == 0:
        print "Block Number %d" % blkno
```

There is no error in the output.

Rev/Undo file is like an onion, you peel it off layer by layer. This example shows that the first byte of the undo data gives a count of the transactions in the block, minus 1.

The undo block is read and the undoblock variable stores the undo data. The first check is on the length of the undo block data which must be larger than 2. In this if statement, the first byte is read in a variable called first. The ord function converts this byte into a number. This value is the number of transactions in the block. But one byte can only store 255 transactions. So, instead of using base 128 encoding, Bitcoin checks if the byte is 253. If yes, then the following 2 bytes store the number.

It is now assumed that there will never be a case where there will be more than 65535 transactions in a block. However, if you want to play by the rules check the next 4 and 8 bytes where the first byte has a value of 254 and 255.

So, whenever the first byte has a value of 253, the next two bytes are read, the first byte is multiplied by 1 and the second by 256. This total is the new value of the variable, first.

The value in the first variable will always be one less than the number of transactions, notran, which we stored in the blockhash table.

A block normally contains dozens of transactions and hence this method is applied. There are no errors in the output. Don't have to call Houston. We could have used our earlier code to read a variable int from disk, but decided not to.

The Second Field in the Undo Block is all About the Number of Inputs

```
ch2609.py
import leveldb
import pycpg2
from cfuncs import *
import subprocess
import json
def reversehash(hash):
```

```

        return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def index(ans):
    (nVersion , offset) = base128(ans , 0)
    (nHeight, offset) = base128(ans , offset)
    (nStatus, offset) = base128(ans, offset)
    (nTx, offset) = base128(ans, offset)
    (nFile, offset) = base128(ans, offset)
    (nDataPos, offset) = base128(ans, offset)
    (nUndoPos, offset) = base128(ans, offset)
    return (nFile , nDataPos, nUndoPos)

def doutput(f , i):
    bitcoinvalue = r8bytes(f)
    outputscriptlen = rvarint(f)
    scriptpubkey = rbytes(f , outputscriptlen).encode('hex')

def dinput(f , i):
    prevtrhash = rhash(f).encode('hex')
    outputindex = rint(f)
    inputscripflen = rvarint(f)
    sigpubkey = rbytes(f , inputscripflen).encode('hex')
    seqno = rint(f)
    return (reversehash(prevtrhash) , outputindex)

def inputs(nFile, nDataPos):
    f = open("/Users/vijaymukhi/Library/Application
    Support/Bitcoin/blocks/blk%05d.dat" % nFile, "rb")
    f.seek(nDataPos , 0)
    header = f.read(80)
    notran = rvarint(f)
    #print "Number of transactions are %d" % notran
    for tno in range ( 0 , notran):
        tver = rint(f)
        noinputs = rvarint(f)
        if tno == 1:
            return noinputs
        for i in range ( 0 , noinputs):
            (hash , index) = dinput(f , i)

```

```
nooutputs = rvarint(f)
for i in range ( 0 , nooutputs):
    doutput(f , i)
locktime = rint(f)
f.close()

def readint(undoblock , offset):
    first = ord(undoblock[offset:offset + 1])
    if first < 253:
        return (first , offset + 1)
    elif first == 253:
        a1 = ord(undoblock[offset + 1:offset + 2])
        a2 = ord(undoblock[offset + 2:offset + 3])
        first = a1 * 1 + a2 * 256
        return (first , offset + 3)

conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index')

for blkno in range (1 , 500000):
    s1 = "Select hash from blockhash where blkno = %d" % (blkno )
    cur.execute(s1)
    row = cur.fetchone()
    hash = row[0]
    hash = reversehash(hash)
    hashb = "62" + hash
    hashb = hashb.decode('hex')
    try:
        ans = db.Get(hashb)
    except:
        print "All over"
        exit()
    (nFile , nDataPos , nUndoPos) = index(ans)
    actualv = inputs(nFile , nDataPos)
    fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/rev%05d.dat" % nFile
    fr = open(fname , "rb")
    fr.seek(nUndoPos , 0)
    undoblock = rbytes(fr , 6)
    fr.close()
    if ord(undoblock[0:1]) != 0:
        (first , offset) = readint(undoblock , 0)
        (second , offset) = readint(undoblock , offset)
        if second == actualv:
            #print "Block Number is %d file number %d actualv %d first %d second %d" %
            (blkno , nFile , actualv , first , second)
        pass
    else:
```



```

print "Houston we have a problem blockno %d first %d second %d actual %d" %
(blkno , first , second , actualv)
exit()
if blkno % 10000 == 0:
print "Block Number %d" % blkno

```

Output

Block Number 450000

All over

Once again, no errors in the output

Let's now understand what the second byte in the undo block represents. As the UTXO set was all about outputs which are unspent, the rev files are all about inputs that will soon be a spent force. The first field holds a count of transactions present in the block. The second field refers to the number of inputs present in the first transaction only.

Let's explain this field using code and not words.

First the block hash is read from the blockhash table. Then we read the file number and the data position of the undo block for a block number using the index function, as before. This function also now returns the starting position of the block data in the blk file. When the Get function of the leveldb database throws an exception, it signifies that the block hash was not present in the index folder which should not happen.

Things change now. A function called inputs is called to display the transactions in a block. It is a copy of code from an earlier program, in the second chapter. This function is first given the block file number saved in the nFile variable and then Data position nDataPos in the blk file. In the case of the undo rev file, this value is 8 bytes from the start of the block. You can bring in the regular error check wherein you move 8 bytes back using the seek function and check for a magic number. We have avoided this check.

The file pointer is at the start of the 80-byte block header, so read these 80 block header bytes and then simply ignore them. Then there is a read on the number of transactions in the block, which is and will always be a number larger or equal to 1. The value is stored in the notran variable.

In the for loop, first is the version byte, completely ignored and then is a number, representing the number of inputs in the current transaction.

A transaction contains a count of the variable number of inputs, followed by the inputs itself, followed by the number of outputs and the outputs themselves. Or focus is on the inputs in the second transaction. The first transaction is the Coinbase transaction, so it is ignored.

First, the inputs and outputs of the first transaction are read and then the number of inputs of the second transaction are read. Thereafter, we simply return from the function. The value returned is the number of inputs of the second transactions. The for statement can not be avoided here as our interest is only in the input of the second transaction. We ignore the Coinbase and all the other transactions as well. Nevertheless, the Coinbase transaction must be read in full as the inputs and outputs of each transaction are stored back to back. On paper, a block has multiple transactions, each one having multiple inputs and outputs.

There are about half a million blocks so the files will be opened and closed a million times. This does get very time consuming. So, for the first time in this book, we are closing a file using the close function, something that we never did earlier. Please believe us, it did not matter initially, but after some time our code effectively stopped working. Therefore, at the end of the inputs function, the blk file is closed and at the end of one block, the rev file is closed. We are returning the number of inputs, noinputs found in the transaction.

Our code however, is still very sloppy. If there is only one transaction in a block, the function returns a None. The inputs function must not be called if there is only one transaction in the block. An if statement could have handled this eventuality. No damage done as the return value stored in variable actualv is used only when there is some data in the undo block.

The first 8 bytes for the magic number and size are ignored as their values are insignificant for now. Instead of reading the entire undo block which can get very large, only 6 bytes are read from the undo block. Also, the first byte of the undo data is checked for a 0, this happens only when we have no undo data.

The first data item in the undo data takes up one byte only if the value is less than 252. A function called readint is introduced that takes the starting bytes and an offset into these bytes, aka the base128 function. If the first byte is a number smaller or equal to 252, this one byte is returned and one byte is added to the offset variable, as a single byte is used up.

If the first byte is 253, the next two bytes are read, the first byte is multiplied by 1 and the second by 256. This sum value is returned and the offset variable is added by 3 since 3 bytes are utilized. You can add the other two use cases also.

The second byte of the undo structure determines the number of inputs in the second transaction of the block.

Block Number is 92951 file number 0 actualv 3 first 216 second 3

Block Number is 92952 file number 0 actualv 1 first 218 second 1

Uncomment the print statement and let's look at the values of block number 92951. The undoblock shows the second transaction having 3 inputs in the variable second. When the number of inputs are calculated using the inputs function, we also get a value of 3. The variable actualv returns the same value. Our understanding of the second byte is to the point as no errors are thrown.

We should not forget that segregated witnesses have spoilt our party. None of what we have said applies from block number 481824 or starting date 24th August 2017.

Understanding the Format of the Undo Files Using Block 170

```
ch2610.py
from cfuncs import *
fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/rev000000.dat"
fr = open(fname , "rb")
for blkno in range(1, 200 ):
    magic = rint(fr)
    sizer = rint(fr)
    undoblock = rbytes(fr , sizer)
    revhasho = rbytes(fr,32)
    if blkno == 170:
        print undoblock.encode('hex')
```

Output

01011301320511db93e1dcdb8a016b49840f8c53bc1eb68a382e97b1482ecad7b148a6909a5c

This is one of the smallest programs but an important one. The file rev000000.dat is opened and scanned through the undo record of every block. The initial blockchain blocks have only 1 transaction, the Coinbase transaction and therefore there is an empty undo block data.

The first block that has undo data or that carries 2 transactions is block number 170. The bytes of the undo data are displayed and then the program exits. The same code will be used in future to display other undo blocks, one at a time.

The idea is to understand the undo block format, one byte at a time. Here we go.

01 This is the count of transactions plus 1 in a block. In our case 2 transactions. Done before.

01 This byte is a count of inputs in the first transaction. Done before.

13 The value 19 is a little tricky to understand. The height of the block is multiplied by 2 to create this number. In the world of odd and even numbers, when an odd or even number is multiplied by 2, it always gives an even number. This technique is useful to create an undo format that saves space on disk. Efficiency is what the doctor asked for.

In other words, this byte indicates two things, the height of the block and the transaction being a Coinbase transaction or not. One is added to the height after multiplying by 2 to create this number. If the transaction is a Coinbase transaction, the height is kept the same as if it is a normal transaction.

If the height is divided by 2 (ignore the decimal place), the original height is obtained. If the height is an odd number, the transaction is a Coinbase transaction. Also, the first bit is not used in the height computation, this bit if on, states the transaction is Coinbase, 0 it is not.

This illustrates that the height of the transaction is 9 and it is a Coinbase transaction. $19 / 2 = 9$ plus 1. Why block 9?

The second transaction starting with f418 has an output that references Bitcoin address:

12cbQLTFMXRnSzktFkuoG3eHoMeFtpTu3S.

This output in turn comes from a transaction.:

0437cd7f8525ceed2324359c2d0ba26006d92d856a9c20fa0241106ee5a597c9.

This transaction hash in turn belongs to block 9 and is a Coinbase transaction.

More on this later

01 Ignore this value as it is a version.

32 It is 50 in decimal which is the amount in the transaction: 50 Bitcoins or BTC.

0511db93e1dcd8a016b49840f8c53bc1eb68a382e97b1482ecad7b148a6909a5c

These 33 bytes of 66 characters can only be an abridged form or the compressed form of the public key.

Block 9 has only one transaction, a Coinbase transaction. Clicking on the transaction hash 0437 shows that the public key starts with 04 and not 05, but the rest of the values are the same.

This output has been marked as spent, thus implying that it will not occur in the UTXO set now.

The million-dollar question: We are deciphering the undo block data of block 170. Where does the block 9 come from?

In the earlier program, in the if statement, replace the 170 to 187.

if blkno == 187:

The output is as follows.

0101000905baa9d36653155627c740b3409a734d4eaf5dcca9fb4f736622ee18efcf0aec2b

0101: The first two bytes are the same as before 01 and 01

00 : The height is 0 and not a block number. This is very odd. As the height is 0, there is no Coinbase and no version field. The block number 0 is the genesis block that will never ever be transferred.

09 : This is not the Bitcoin value 9 BTC as this amount is compressed, more like 100000000.

There is a reason why block 187 is so different from block 170. The next program explains it.

05baa9d36653155627c740b3409a734d4eaf5dcca9fb4f736622ee18efcf0aec2b

This is obviously a public key.

Understanding the Undo Block Data for Two Blocks: 170 and 187

```
ch2611.py
import leveldb
import psycpg2
from cfuncs import *
import subprocess
import json
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)
def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
```

```

    offset = offset + 1
    n = (n << 7) | (chData & 0x7F)
    if chData & 0x80 == 128:
        n = n + 1
    else:
        return (n, offset)

def index(ans):
    (nVersion , offset) = base128(ans , 0)
    (nHeight, offset) = base128(ans , offset)
    (nStatus, offset) = base128(ans, offset)
    (nTx, offset) = base128(ans, offset)
    (nFile, offset) = base128(ans, offset)
    (nDataPos, offset) = base128(ans, offset)
    (nUndoPos, offset) = base128(ans, offset)
    return (nFile , nDataPos, nUndoPos)

def doutput(f , i):
    bitcoinvalue = r8bytes(f)
    outputscripplen = rvarint(f)
    scriptpubkey = rbytes(f , outputscripplen).encode('hex')

def dinput(f , i):
    prevtrhash = rhash(f).encode('hex')
    outputindex = rint(f)
    inputscripplen = rvarint(f)
    sigpubkey = rbytes(f , inputscripplen).encode('hex')
    seqno = rint(f)
    return (reversehash(prevtrhash) , outputindex)

def inputs(nFile, nDataPos):
    inputlist = []
    vnoinputs = 0
    f = open("/Users/vijaymukhi/Library/Application
    Support/Bitcoin/blocks/blk%05d.dat" % nFile, "rb")
    f.seek(nDataPos , 0)
    header = f.read(80)
    notran = rvarint(f)
    for tno in range ( 0 , notran):
        tver = rint(f)
        noinputs = rvarint(f)
        if tno == 1:
            vnoinputs = noinputs
        for i in range ( 0 , noinputs):
            (hash , index) = dinput(f , i)
            inputlist.append((hash , index ))
        nooutputs = rvarint(f)
        for i in range ( 0 , nooutputs):
            doutput(f , i)

```

```
    locktime = rint(f)
    return (inputlist, vnoinputs , notran)

def readint(undoblock , offset):
    first = ord(undoblock[offset:offset + 1])
    if first < 253:
        return (first , offset + 1)
    elif first == 253:
        a1 = ord(undoblock[offset + 1:offset + 2])
        a2 = ord(undoblock[offset + 2:offset + 3])
        first = a1 * 1 + a2 * 256
        return (first , offset + 3)

conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost'
password='accord'")
cur = conn.cursor()
db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index')

for blkno in [170 , 187]:
    print "Block Number %d" % (blkno )
    s1 = "Select hash from blockhash where blkno = %d" % (blkno )
    cur.execute(s1)
    row = cur.fetchone()
    hash = row[0]
    hash = reversehash(hash)
    hashb = "62" + hash
    hashb = hashb.decode('hex')
    try:
        ans = db.Get(hashb)
    except:
        print "Over and out"
        break
    (nFile , nDataPos , nUndoPos) = index(ans)
    (inputlist , vnoinputs , notran) = inputs(nFile , nDataPos)
    fname = "/Users/vijaymukhi/Library/Application
Support/Bitcoin/blocks/rev%05d.dat" % nFile
    fr = open(fname , "rb")
    fr.seek(nUndoPos - 8 , 0)
    magic = rint(fr)
    sizer = rint(fr)
    undoblock = rbytes(fr , sizer)
    undohash = rbytes(fr , 32)
    fr.close()
    if sizer > 1:
        (first , offset) = readint(undoblock , 0)
        (second , offset) = readint(undoblock , offset)
        for undoindex in range (1 , len(inputlist)):
            print "Undo Bytes %s Length is %d" %
            (undoblock[offset:].encode('hex') , len(undoblock[offset:].encode('hex')))
```

```

oldoffset = offset
(nCode , offset) = base128(undoblock , offset)
print "nCode %d oldoffset %d offset %d " % (nCode , oldoffset , offset )
nHeight = nCode / 2
fCoinBase = nCode & 1
print "first %d second %d nHeight %d fCoinbase %d" % (first , second ,
nHeight , fCoinBase)
if nHeight > 0 :
oldoffset = offset
(version , offset) = base128(undoblock , offset)
print "nVersion %d offset old %d new %d " % (version , oldoffset , offset )
thash = inputlist[undoindex][0]
oindex = inputlist[undoindex][1]
print "No of transaction %d No of inputs %d" % (notran , vnoinputs)
print "Transaction Hash %s:%d" % (thash , oindex)
raw = subprocess.check_output(["bitcoin-cli" , "getrawtransaction" , thash , "1" ])
raw = json.loads(raw)
bvalue = raw["vout"][oindex]["value"]
otype = raw["vout"][oindex]["scriptPubKey"]["type"]
print "Bitcoin Value bitcoin-cli %f" % bvalue
print "Type of transaction bitcoin-cli %s" % otype
oldoffset = offset
(amount, offset) = base128a(undoblock , offset)
print "Undo Amount %d oldoffset %d offset %d " % (amount, oldoffset , offset )
pubkey1 = raw["vout"][oindex]["scriptPubKey"]["hex"]
if otype == 'pubkey':
undooffset = 33
pubkey = undoblock[offset + 1: offset + undooffset]
pubkey1 = pubkey[4:68]
print "Public Key Undo File %s" % pubkey.encode('hex')
print "Public Key bitcoin-cli %s" % pubkey1
print "nheight is %d" % nHeight
if nHeight > 0:
bhash = raw['block hash']
raw1 = subprocess.check_output(["bitcoin-cli" , "getblock" , bhash])
raw1 = json.loads(raw1)
nHeight1 = raw1["height"]
bnheight = nHeight == nHeight1
else:
nHeight1 = 0
bnheight = True
print "height is %d" % nHeight1
bfirst = first == notran - 1
bsecond = second == vnoinputs
bamount = amount / 100000000 == int(bvalue)
bpubkey = pubkey.encode('hex') == str(pubkey1)

```

```
print "first %s second %s height %s amount %s public key %s " %
(bfirst , bsecond , bnheight , bamount , bpubkey )
if bfirst and bsecond and bamount and bpubkey and bnheight:
print "Rev Data matches for block number %d" % blkno
else:
print "Vijay Mukhi is a embecile at block number %d" % blkno
exit()
print
```

Output

Block Number 170

Undo Bytes

1301320511db93e1dcd8a016b49840f8c53bc1eb68a382e97b1482ecad7b148a6909a5c

Length is 72

nCode 19 oldoffset 2 offset 3

first 1 second 1 nHeight 9 fCoinbase 1

nVersion 1 offset old 3 new 4

No of transaction 2 No of inputs 1

Transaction Hash

0437cd7f8525ceed2324359c2d0ba26006d92d856a9c20fa0241106ee5a597c9:0

Bitcoin Value bitcoin-cli 50.000000

Type of transaction bitcoin-cli pubkey

Undo Amount 5000000000 oldoffset 4 offset 5

Public Key Undo File

11db93e1dcd8a016b49840f8c53bc1eb68a382e97b1482ecad7b148a6909a5c

Public Key bitcoin-cli

11db93e1dcd8a016b49840f8c53bc1eb68a382e97b1482ecad7b148a6909a5c

nheight is 9

height is 9

first True second True height True amount True public key True

Rev Data matches for block number 170

Block Number 187

Undo Bytes

000905baa9d36653155627c740b3409a734d4eaf5dcca9fb4f736622ee18efcf0aec2b

Length is 70

nCode 0 oldoffset 2 offset 3

first 1 second 1 nHeight 0 fCoinbase 0

No of transaction 2 No of inputs 1

Transaction Hash

12b5633bad1f9c167d523ad1aa1947b2732a865bf5414eab2f9e5ae5d5c191ba:0

Bitcoin Value bitcoin-cli 1.000000

Type of transaction bitcoin-cli pubkey

Undo Amount 100000000 oldoffset 3 offset 4

Public Key Undo File

baa9d36653155627c740b3409a734d4eaf5dcca9fb4f736622ee18efcf0aec2b

Public Key bitcoin-cli


```

baa9d36653155627c740b3409a734d4eaf5dcca9fb4f736622ee18efcf0aec2b
nheight is 0
height is 0
first True second True height True amount True public key True
Rev Data matches for block number 187

```

Let's now decipher the undo data in the problematic blocks, 170 and 187. The for statement loops twice, as we handle the blkno variable having a value of 170 and 187. A very long explanation, the heart and soul of this chapter!

Some theory first, the undo data is written when the Bitcoin blocks are downloaded. Simultaneously, the chainstate leveldb undo key-value pairs also get updated. So, if a block contains 2 transactions, the first one always is the Coinbase transaction; there is no undo data written for it. The input of the second transaction may have one or more transaction hashes that point to an output that brings in the money or Bitcoins to be transferred. Same statement made umpteen times before.

Now, to one of the most important lines in this chapter. The transaction hash value and the output index referred by the current input must be present in the UTXO data set. However, after and the key is after, processing this block, this transaction hash and index will be removed from the UTXO set because its Bitcoins have been just been spent. The key again is before and after. To sum up multiple times, before a block is confirmed, the outputs the inputs point to have to be unspent. The nanosecond the block is confirmed, these outputs are spent.

Just before the transaction is cleared by the miners, the output referred to, by the input must be present in the UTXO set. The second it is mined, the output status changes from unspent to spent. It is booted out from the UTXO set on your machine and every other machine in the galaxy.

The UTXO set contains only unspent outputs. This set expands and contracts with every block. Transaction outputs get spent and fresh outputs are added. Its contents are dynamic. As the number of blocks bring in more transactions, the UTXO set also grows. These transactions have inputs so, the UTXO set also shrinks.

The last entity that gets affected, is the index folder where one key-value pair is added for each block and for every transaction, if transaction indexing is on. Indexing is required as we have no idea where a block starts in the rev or blk files.

To understand rev file undo data, the reindexing of the Bitcoin blockchain plays a very significant role. Bitcoin deletes all the key-value pairs in the chainstate and index folders when reindexing. The only way to reindex a blockchain is by passing -reindex as a command line argument to bitcoind or by modifying the source code.

Bitcoin also deletes the rev files one by one and starts creating them from scratch. It first deletes the rev file rev00000.dat, writes the undo blocks from the beginning. It then deletes the file rev00001.dat and so on. The blk files are not deleted as they take too long to be downloaded from the Internet. It also assumes that the data in these block files are not tampered.

Some more revision.

The UTXO set is a very dynamic key-value database. When bitcoind creates the undo data for block 170, the UTXO data set has one set of records. After it finishes writing the undo data, it also updates the UTXO set. All this is normally done when a block is added to the blockchain.

All inputs that are present in the transactions have their outputs removed from the UTXO set as these outputs have been spent by transactions in this block. At the same time, the newly created outputs have money in them, so these are added to the UTXO set. Some outputs are removed from the UTXO set, some outputs are added after processing a block. We keep repeating hoping we are on the same page with you.

Back to the code. Instead of chugging through half a million key value pairs, we are only looping through two blocks 170 and 187. First, their block hashes are retrieved from the blockhash table.

It is difficult to guess where the block data and undo data will be found in the blk and rev files. Therefore, the hash is reversed and 62 is added to retrieve the value for this key from the index leveldb folder. Standard stuff!

The index folder is as always called index. The leveldb database is single user. The minute a program accesses the folder, it creates a file called LOCK with 0 bytes. As these programs take days to run, we simply create a duplicate of the folder and delete the LOCK file. What comes to us after the call of the index function, is the file number and positions in the rev and blk where the block and undo data for this block number starts. All this is done by a scaled down version of the index function extracted from the index chapter.

The inputs function is also taken from the chapter on transactions. Here, lots of data is returned, 3 parameters in all, 1 list and two numbers. The focus here is on inputs that point to outputs. As a repetition, the list of inputs is stored back to back and are preceded with the number of inputs for those transactions. The same structure is used for outputs which is ignored.

In the inputs function, the count of inputs is stored in the vnoinputs variable. This applies only to the second transaction or the first non-Coinbase transaction. The value in the vnoinputs variable is returned. It is our way of saying that we are interested only in the first transactions inputs only. We have not yet explained why the second transaction and not the third. This value is however now not returned in the if statement. In the inputlist list, only the hash value and the output index of all the inputs in this transaction supplying bitcoins is added. This is different as earlier we returned all the inputs. The number of transactions in the block is also returned for future use. This value is stored in the notran variable.

The inputs function is also passed the blk file number and the start position of the block in the file. As before, the 8 bytes for the magic bytes and the size are ignored. The block header is read and the transaction data is analyzed in the for loop. This ends the task of the inputs function.

Using the nFile variable which uniquely identifies the file, the corresponding rev file is opened. The file pointer is positioned to the beginning of the rev file data by seeking back 8 bytes. The magic bytes are ignored but the size of the undo data is stored in the variable sizer. Using this variable, the undo data is read from the file. For efficiency purposes, the blk and rev files are closed. If you were being observant, we are only closing the rev file and not the blk file as we are only working at 50% efficiency. Bad joke to cover up a bug in our code.

The undo hash data, though read is not used now. There are approximately 80,000 blocks whose undo data size is 1. This divulges the fact that these blocks only have a single transaction which is a Coinbase transaction. Nothing can be done here as the undo data value is a 0.

The if statement checks the size of the undo block stored in variable size; it must be larger than 1, just to ensure that the empty undo blocks are ignored. These optimizations are meant more for later programs when they are executed on all the rev file data.

The readint function reads the first and second byte of the undoblock. In this case, their values will be 1 and 1. As before, the first variable stores the first field and the second variable stores the second field. These values are displayed later in our code.

The first variable's value of 1 signifies that the block has only 2 (1 + 1) transactions and the second transaction has only 1 input. These values are checked at the end of the code, not manually, but by using code.

Now all the transactions in the block are scanned. In our case, there are only 2 transactions. The for loop is not needed now as it loops only once. The variable undoindex starts with 1 and not 0, to ignore the first Coinbase transaction. The size of the inputlist list is 2 and not 1 because there are two inputs. This fact is confirmed in our blockchain explorer.

The Coinbase transaction can be avoided in the inputs function but we leave it as is. The inputlist list stores only the

inputs of all the transactions in the block. Thus, after the inputs of transaction 10 are the inputs of transaction 11. A block can have multiple transactions which in turn have multiple inputs. Once again, with the current data, the for loop will loop only once. Technically not needed.

The base128 function reads the block height or the nCode variable, which is stored next. This also helps in deciphering the number of bytes used to store this value. The offset variable is used to read the undo block data. This variable is used to read every field.

The value of the offset variable is stored in the oldoffset variable. Then the base128 function is called. Here, we display the new and the old value of the offset variable. The value of the variable nCode is 19 and the pointer now moves just one byte. We do this only to figure out how many bytes a certain field has used up.

Now to obtain the height of the block. The height of the block is nCode/2, which gives a value of 9. This is a simple division, ignore the decimal places if any. This nCode variable also determines if it is a Coinbase transaction, by reading the first bit. As the first bit is on, the fCoinbase variable is 1.

Since the height is larger than 0, the next byte is the version of the transaction which is 1 in this case.

The next undo block has a height of 0 and thus no version byte. This is rationale of the if statement. No height, no version. The oldoffset variable is used only to check if the right number of bytes are read.

This is what we know about block 170.

```
input 12cbQLTFMXRnSzktFkuoG3eHoMeFtpTu3S
```

This transaction hash comes from block 9 and it also has an output, the above Bitcoin address.

```
0437cd7f8525ceed2324359c2d0ba26006d92d856a9c20fa0241106ee5a597c9.
```

It is a Coinbase transaction. Therefore, the fCoinbase was 1.

On the other hand, the block 187 has an input address:

```
13HtsYzne8xVPdGDnmJX8gHgBZerAfJGEf. It references the transaction :
12b5633bad1f9c167d523ad1aa1947b2732a865bf5414eab2f9e5ae5d5c191ba.
```

The transaction is in block no. 183. It is not a Coinbase transaction but the second transaction in the block. We chose two different types of rev blocks.

The block no. 170 has two transactions. The one we are interested in has one input only. We read the transaction hash and the output index of the output (sender of Bitcoins) using the inputlist list with loop variable undoindex as an offset into the list. The hash and output index in turn are the first and second members of this list.

The loop variable undoindex is used in cases where there are more than 2 inputs in the inputlist list. Once again, the transaction hash is the first member of the list and the output index the second member. These values are stored in variable, thash and oindex.

In the website blockchain.info, let's examine the transaction beginning with 0437. Please read the output for the entire hash value. It clearly shows that the transaction starting with 0437 belongs to block 9 and the first and the only output has a hash value starting with 11db93. The first byte is 04 and not 05.

Therefore, the height in the undoblock data is 9. The height is not of the current block where the transaction resides but it is the height in which the transaction that supplies outputs are present. A manual check for every transaction gets a bit tedious.

The client program bitcoin-cli is called with the getrawtransaction command and this transaction hash value. The last parameter asks for a json formatted verbose output instead of a series of bytes.

The raw output is converted into a Python dictionary and three fields are read from it. The first field is the output bitcoin value and it is stored in a field `vout`, which is a list. The `oindex` index variable accesses this list and then the field called `value` facilitates in recovering just the value in Bitcoins. The Bitcoin value is 50.000 and is stored in the `bvalue` variable.

The second field is a type or the type field of a field called `scriptPubKey`. In our case, it is `pubkey`.

The `base128a` function reads the amount that follows in the undo block. The amounts including the undo block are compressed. Like the UTXO set, the amount is 5000000000, at one level both are the same

The UTXO chapter has the explanation for the different forms of public key types. Here, the type of transaction is checked to be `pubkey`, if yes, the next 33 bytes are read in the variable `pubkey`. The variable `pubkey` refers to the public key stored in the undo data. The undo bytes start with a 05 and it is ignored. Therefore, the bytes are read from offset + 1 and not offset. Also, only 33 bytes or the value stored in the `undooffset` variable are read and not all bytes till the end of the string. It may work in this case, but it will not when dealing with multiple inputs in the undo data.

The value of `pubkey` variable is decoded, whereas the value of variable `pubkey1` is encoded. The variable `pubkey1` is generated by the `getrawtransaction` command. This command gives the public key where the first size byte is 0x41 and the second byte is 04, but they are ignored.

Both the undo data public key and the value received from the `getrawtransaction` command are displayed. They have the same value.

Now let's confirm if we have the right height in the undo data. The first check is on the height stored in the undo block. It must be larger than 0. The `getrawtransaction` method's output variable, `raw` has a member `blockhash`. This member retrieves the block hash of the block having the transaction starting with 0437. Having got the block hash, the `getblock` command is used to return the block details. The height field is extracted from these details. Now we check the two heights, `nHeight` of the undo block and `nHeight1` returned by our client `bitcoin-cli`. They have the same value.

If the height is 0, then the variable `bnheight` is set to `True`.

For rest of the Boolean checks, make sure that the undo block data is good with no errors. The variable `first` is the number of transactions in the block minus 1. The variable returned from the `inputs` function, `notran` gives a count on the transactions in the block. If both variables have the same value, the variable `bfirst` is set to `true`.

The second variable `bsecond` checks if the second variable is the same as the number of inputs returned by the `inputs` function, `vnoinputs`. This count is the number of inputs of the second transaction only and not a running count of all inputs of all transactions in the block.

The `bamount` variable is true if the amounts are the same. As seen earlier, the `bvalue` variable from the `getrawtransaction` command is a number with decimal places and the value in the undo block is an integer stored in the `amount` variable. We cannot compare a decimal number to a floating-point number, so the `bitcoin-cli` returned value is converted to a number using the `int` function and the undo block number is divided by 1 with 8 0's.

The value of `pubkey1` is obtained from the `bitcoin-cli` program. The `pubkey` variable represents a public key in the `undoblock` string and it is compared with variable `pubkey1`.

Finally, the heights are compared; variable `nHeight` is the `undoblock` height and `nHeight1` is the height from the command `getblock`.

These five boolean variables are checked. If they have a value of false, then the undo data has some errors or we have not understood the undo data format completely, which is more likely the case.

Let's now understand the undo data of block 187. The height variable or better still variable `nCode` is 0. The only way to locate the block holding this transaction is by looking up transaction hash value starting with 12b5, in this case only in `blockchain.info`. The output confirms that the transaction is in block number 183. In these cases, the `bnheight` variable is set to `True`.

According to the comments in the Bitcoin source the height, Coinbase and version fields are called metadata. The height and Coinbase are treated as one entity and the version is another entity. If the height variable is 0, the version field is not written. In other words, if the height variable has a value, the version field is written and then and only then we read it.

The metadata is written only when it is the last unspent output. It is impossible to find out the contents of the UTXO set when creating the undo block.

Handling Some More Undo Blocks

```

ch2612.py
import leveldb
import psycopg2
from cfuncs import *
import subprocess
import json
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)
def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n
def base128(ans , offset):

```

```
n = 0
while True:
    chData = ord(ans[offset:offset + 1])
    offset = offset + 1
    n = (n << 7) | (chData & 0x7F)
    if chData & 0x80 == 128:
        n = n + 1
    else:
        return (n,offset)

def index(ans):
    (nVersion , offset) = base128(ans , 0)
    (nHeight, offset) = base128(ans , offset)
    (nStatus, offset) = base128(ans, offset)
    (nTx, offset) = base128(ans, offset)
    (nFile, offset) = base128(ans, offset)
    (nDataPos, offset) = base128(ans, offset)
    (nUndoPos, offset) = base128(ans, offset)
    return (nFile , nDataPos, nUndoPos)

def doutput(f , i):
    bitcoinvalue = r8bytes(f)
    outputscripflen = rvarint(f)
    scriptpubkey = rbytes(f , outputscripflen).encode('hex')

def dinput(f , i):
    prevtranhsh = rhash(f).encode('hex')
    outputindex = rint(f)
    inputscripflen = rvarint(f)
    sigpubkey = rbytes(f , inputscripflen).encode('hex')
    seqno = rint(f)
    return (reversehash(prevtranhsh) , outputindex)

def inputs(nFile, nDataPos):
    inputlist = []
    vnoinputs = 0
    f = open("/Users/vijaymukhi/Library/Application
Support/Bitcoin/blocks/blk%05d.dat" % nFile, "rb")
    f.seek(nDataPos , 0)
    header = f.read(80)
    notran = rvarint(f)
    for tno in range ( 0 , notran):
        tver = rint(f)
        noinputs = rvarint(f)
        if tno == 1:
            vnoinputs = noinputs
        for i in range ( 0 , noinputs):
            (hash , index) = dinput(f , i)
            inputlist.append((hash , index ))
```

```

nooutputs = rvarint(f)
for i in range ( 0 , nooutputs):
    doutput(f , i)
    locktime = rint(f)
    return (inputlist, vnoinputs , notran)

def readint(undoblock , offset):
    first = ord(undoblock[offset:offset + 1])
    if first < 253:
        return (first , offset + 1)
    elif first == 253:
        a1 = ord(undoblock[offset + 1:offset + 2])
        a2 = ord(undoblock[offset + 2:offset + 3])
        first = a1 * 1 + a2 * 256
        return (first , offset + 3)

conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost'
password='accord'")
cur = conn.cursor()
db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index')

for blkno in range(1, 500000):
    if blkno % 1000 == 0:
        print "Block Number%d" % (blkno )
        s1 = "Select hash from blockhash where blkno = %d" % (blkno )
        cur.execute(s1)
        row = cur.fetchone()
        hash = row[0]
        hash = reverseahash(hash)
        hashb = "62" + hash
        hashb = hashb.decode('hex')
        try:
            ans = db.Get(hashb)
        except:
            print "Over and out"
            break
        (nFile , nDataPos , nUndoPos) = index(ans)
        fname = "/Users/vijaymukhi/Library/Application
Support/Bitcoin/blocks/rev%05d.dat" % nFile
        fr = open(fname , "rb")
        fr.seek(nUndoPos - 8 , 0)
        magic = rint(fr)
        sizer = rint(fr)
        undoblock = rbytes(fr , sizer)
        undohash = rbytes(fr , 32)
        fr.close()
        if sizer > 1:
            (first , offset) = readint(undoblock , 0)

```

```
(second , offset) = readint(undoblock , offset)
if first == 1 and second >= 1:
    (inputlist , vnoinputs , notran) = inputs(nFile , nDataPos)
    for undoindex in range (1 , len(inputlist)):
        #print "Undo Bytes %s Length is %d" % (undoblock[offset:].encode('hex') ,
        len(undoblock[offset:].encode('hex')))
        oldoffset = offset
        (nCode , offset) = base128(undoblock , offset)
        #print "nCode %d oldoffset %d offset %d " % (nCode , oldoffset , offset )
        nHeight = nCode / 2
        fCoinBase = nCode & 1
        #print "first %d second %d nHeight %d fCoinbase %d" % (first , second ,
        nHeight , fCoinBase)
        if nHeight > 0 :
            oldoffset = offset
            (version , offset) = base128(undoblock , offset)
            #print "nVersion %d offset old %d new %d " % (version , oldoffset , offset )
            thash = inputlist[undoindex][0]
            oindex = inputlist[undoindex][1]
            #print "No of transaction %d No of inputs %d" % (notran , vnoinputs)
            #print "Transaction Hash %s:%d" % (thash , oindex)
            raw = subprocess.check_output(["bitcoin-cli" , "getrawtransaction" , thash , "1" ])
            raw = json.loads(raw)
            bvalue = raw["vout"][oindex]["value"]
            otype = raw["vout"][oindex]["scriptPubKey"]["type"]
            #print "Bitcoin Value bitcoin-cli %f" % bvalue
            #print "Type of transaction bitcoin-cli %s" % otype
            oldoffset = offset
            (amount, offset) = base128a(undoblock , offset)
            #print "Undo Amount %d oldoffset %d offset %d " % (amount, oldoffset , offset )
            pubkey1 = raw["vout"][oindex]["scriptPubKey"]["hex"]
            if otype == 'nonstandard':
                break
            elif otype == 'pubkey':
                undooffset = 33
                pubkey = undoblock[offset + 1: offset + undooffset]
                pubkey1 = pubkey[4:68]
            elif otype == 'pubkeyhash':
                undooffset = 21
                pubkey = undoblock[offset + 1: offset + undooffset]
                pubkey1 = pubkey.decode('hex')
                pubkey1 = pubkey1[3:23]
                pubkey1 = pubkey1.encode('hex')
            elif otype == 'scripthash':
                undooffset = 21
                pubkey = undoblock[offset + 1: offset + undooffset]
```



```

pubkey1 = pubkey1[4:44]
else:
    undooffset = 33
    pubkey = undoblock[offset + 1: offset + undooffset]
    pubkey1 = pubkey1[4:68]
    print "Bailing out"
    print pubkey1
    exit()
#print "Public Key Undo File %s" % pubkey.encode('hex')
#print "Public Key bitcoin-cli %s" % pubkey1
#print "nheight is %d" % nHeight
if nHeight > 0:
    bhash = raw['blockhash']
    raw1 = subprocess.check_output(["bitcoin-cli", "getblock", bhash])
    raw1 = json.loads(raw1)
    nHeight1 = raw1["height"]
    bnheight = nHeight == nHeight1
else:
    nHeight1 = 0
    bnheight = True
#print "height is %d" % nHeight1
bfirst = first == notran - 1
bsecond = second == vnoinputs
bamount = amount / 100000000 == int(bvalue)
bpubkey = pubkey.encode('hex') == str(pubkey1)
#print "first %s second %s height %s amount %s public key %s " % (bfirst ,
    bsecond , bnheight , bamount , bpubkey )
if bfirst and bsecond and bamount and bpubkey and bnheight:
    #print "Rev Data matches for block number %d" % blkno
    pass
else:
    print "Vijay Mukhi is a embecile at block number %d" % blkno
    exit()
offset = offset + undooffset
#print

```

Output

Block Number 489000

Over and out

This program is in continuation of an earlier example. The block to process has its first byte as 1 in the undo block. The second byte must also be greater than or equal to 1. The program does not check all the blocks. Slow and steady wins the race.

The first change is in the for statement. Earlier, only two blocks were checked, now the loop starts from block number 1 and ends at a very large number, about half a million. There is an if statement where the values of the first and second variable are checked to be 1 and ≥ 1 . Not all undo blocks will meet this condition. Its makes our code run faster and reduces the waiting time.

The biggest change here is in checking the type of output. As seen earlier in the UTXO set, a check is performed for four types nonstandard, pubkey, pubkeyhash and scripthash. The multi-signature hash was never ever encountered in this program.

When the type is nonstandard, there is a break as nobody can handle a non-standard output. The rules in the UTXO chapter are followed to handle a pubkeyhash and a scripthash where the public key length is 33 bytes. If there is another hash type, the program quits after printing a message. At the end of the code, multiple undo blocks placed sequentially in the variable undoblock are handled. The variable offset allows us to read the inputs seamlessly.

The offset variable gives the position from where the undo block's fields are to be read. The last field is the public key and the offset variable points to the undo block's public key.

The undooffset variable states the number of bytes to read of the public key. It is set to a different value with a different type of hash type. At the end of the inner for loop, the value in variable undooffset is simply added to the value in variable offset, to read the next undo record. In this manner, the undo structure is read one at a time.

The first and second bytes appear only once and the undo block starts with the height field. Very slow program but not so bad. No errors so far.

Checking all the Undo Blocks

```
ch2613.py
import leveldb
import pycpg2
from cfuncs import *
import subprocess
import json
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)
def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
```

```

    n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def index(ans):
    (nVersion , offset) = base128(ans , 0)
    (nHeight, offset) = base128(ans , offset)
    (nStatus, offset) = base128(ans, offset)
    (nTx, offset) = base128(ans, offset)
    (nFile, offset) = base128(ans, offset)
    (nDataPos, offset) = base128(ans, offset)
    (nUndoPos, offset) = base128(ans, offset)
    return (nFile , nDataPos, nUndoPos)

def doutput(f , i):
    bitcoinvalue = r8bytes(f)
    outputscripflen = rvarint(f)
    scriptpubkey = rbytes(f , outputscripflen).encode('hex')

def dinput(f , i):
    prevtrhash = rhash(f).encode('hex')
    outputindex = rint(f)
    inputscripflen = rvarint(f)
    sigpubkey = rbytes(f , inputscripflen).encode('hex')
    seqno = rint(f)
    return (reverseahash(prevtrhash) , outputindex)

def inputs(nFile, nDataPos):
    inputlist = []
    vnoinputs = 0
    f = open("/Users/vijaymukhi/Library/Application
Support/Bitcoin/blocks/blk%05d.dat" % nFile, "rb")
    f.seek(nDataPos , 0)
    header = f.read(80)

```

```
    notran = rvarint(f)
    for tno in range ( 0 , notran):
        tver = rint(f)
        noinputs = rvarint(f)
        for i in range ( 0 , noinputs):
            (hash , index) = dinput(f , i)
            inputlist.append((hash , index , tno , noinputs))
        nooutputs = rvarint(f)
        for i in range ( 0 , nooutputs):
            doutput(f , i)
        locktime = rint(f)
        return (inputlist, notran)

def readint(undoblock , offset):
    first = ord(undoblock[offset:offset + 1])
    if first < 253:
        return (first , offset + 1)
    elif first == 253:
        a1 = ord(undoblock[offset + 1:offset + 2])
        a2 = ord(undoblock[offset + 2:offset + 3])
        first = a1 * 1 + a2 * 256
        return (first , offset + 3)

conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost'
password='accord'")
cur = conn.cursor()
db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index')

for blkno in range(1, 500000):
    if blkno % 1000 == 0:
        print "Block Number%d" % (blkno )
        s1 = "Select hash from blockhash where blkno = %d" % (blkno )
        cur.execute(s1)
        row = cur.fetchone()
        hash = row[0]
        hash = reversehash(hash)
        hashb = "62" + hash
        hashb = hashb.decode('hex')
        try:
            ans = db.Get(hashb)
        except:
            print "Over and out"
            break
        (nFile , nDataPos , nUndoPos) = index(ans)
        fname = "/Users/vijaymukhi/Library/Application
Support/Bitcoin/blocks/rev%05d.dat" % nFile
        fr = open(fname , "rb")
        fr.seek(nUndoPos - 8 , 0)
```

```

magic = rint(fr)
sizer = rint(fr)
undoblock = rbytes(fr , sizer)
undohash = rbytes(fr , 32)
fr.close()
if sizer > 1:
    (first , offset) = readint(undoblock , 0)
    (second , offset) = readint(undoblock , offset)
    (inputlist , notran) = inputs(nFile , nDataPos)
    for undoindex in range (1 , len(inputlist)):
        #print "Undo Bytes %s Length is %d" % (undoblock[offset:].encode('hex') ,
        len(undoblock[offset:].encode('hex')))
        if inputlist[undoindex][2] != inputlist[undoindex - 1][2] and undoindex >= 2:
            #print "(%d) Transaction change" % blkno
            oldoffset = offset
            (noinputs1 , offset ) = readint(undoblock , offset)
            #print "number of inputs %d notran %d oldoffset %d offset %d" %
            (noinputs1 , inputlist[undoindex][3] , oldoffset , offset )
            if noinputs1 != inputlist[undoindex][3]:
                print "No of inputs in transactions do not match"
                exit()

            oldoffset = offset
            (nCode , offset) = base128(undoblock , offset)
            #print "nCode %d oldoffset %d offset %d " % (nCode , oldoffset , offset )
            nHeight = nCode / 2
            fCoinBase = nCode & 1
            #print "first %d second %d nHeight %d fCoinbase %d" % (first , second ,
            nHeight , fCoinBase)
            if nHeight > 0 :
                oldoffset = offset
                (version , offset) = base128(undoblock , offset)
                #print "nVersion %d offset old %d new %d " % (version , oldoffset , offset )
                thash = inputlist[undoindex][0]
                oindex = inputlist[undoindex][1]
                #print "No of transaction %d No of inputs %d" % (notran , vnoinputs)
                #print "Transaction Hash %s:%d" % (thash , oindex)
                raw = subprocess.check_output(["bitcoin-cli" , "getrawtransaction" , thash , "1" ] )
                raw = json.loads(raw)
                bvalue = raw["vout"][oindex]["value"]
                otype = raw["vout"][oindex]["scriptPubKey"]["type"]
                #print "Bitcoin Value bitcoin-cli %f" % bvalue
                #print "Type of transaction bitcoin-cli %s" % otype
                oldoffset = offset
                (amount, offset) = base128a(undoblock , offset)
                #print "Undo Amount %d oldoffset %d offset %d " % (amount, oldoffset , offset )
                pubkey1 = raw["vout"][oindex]["scriptPubKey"]["hex"]

```

```
if otype == 'nonstandard':
    break
elif otype == 'pubkey':
    undooffset = 33
    pubkey = undoblock[offset + 1: offset + undooffset]
    pubkey1 = pubkey[4:68]
elif otype == 'pubkeyhash':
    undooffset = 21
    pubkey = undoblock[offset + 1: offset + undooffset]
    pubkey1 = pubkey.decode('hex')
    pubkey1 = pubkey1[3:23]
    pubkey1 = pubkey1.encode('hex')
elif otype == 'scripthash':
    undooffset = 21
    pubkey = undoblock[offset + 1: offset + undooffset]
    pubkey1 = pubkey[4:44]
elif otype == 'multisig':
    print "Multisig blocknumber %d %x" % (blkno, ord(undoblock[offset: offset + 1]))
    if ord(undoblock[offset: offset + 1]) == 0x80:
        offset = offset + 1
    if ord(undoblock[offset: offset + 1]) == 0x81:
        offset = offset + 1
    if ord(undoblock[offset: offset + 1]) >= 0x82:
        offset = offset + 1
    undooffset = len(pubkey1) / 2 + 1
    pubkey = undoblock[offset + 1: offset + undooffset]
else:
    undooffset = 33
    pubkey = undoblock[offset + 1: offset + undooffset]
    pubkey1 = pubkey[4:68]
    print "Bailing out"
    print pubkey1
    exit()
#print "Public Key Undo File %s" % pubkey.encode('hex')
#print "Public Key bitcoin-cli %s" % pubkey1
#print "nheight is %d" % nHeight
if nHeight > 0:
    bhash = raw['blockhash']
    raw1 = subprocess.check_output(["bitcoin-cli", "getblock", bhash])
    raw1 = json.loads(raw1)
    nHeight1 = raw1["height"]
    bnheight = nHeight == nHeight1
else:
    nHeight1 = 0
    bnheight = True
#print "height is %d" % nHeight1
bfirst = first == notran - 1
```

```

bsecond = second == inputlist[1][3]
bamount = amount / 100000000 == int(bvalue)
bpubkey = pubkey.encode('hex') == str(pubkey1)
#print "first %s second %s height %s amount %s public key %s " %
(bfirst , bsecond , bnheight , bamount , bpubkey )
if bfirst and bsecond and bamount and bpubkey and bnheight:
#print "Rev Data matches for block number %d" % blkno
pass
else:
print "Vijay Mukhi is a embecile at block number %d" % blkno
exit()
offset = offset + undooffset
#print

```

This program starts from undo block number 1 and then verifies all the undo block data. The first change is in the return values of the inputs function.

First in the inputs function, an empty list called inputlist is created. Then loop goes on for the number of transactions in the block. Transactions are known to have multiple inputs. If a transaction has 5 inputs, the second for statement loops 5 times. Each input in a transaction is added to this inputlist list. Every member has fields.

A better approach would be to create another list that returns the number of inputs per transaction only. If there are 4 transactions, a list of 4 is returned. Each list would contain multiple inputs. Though it is bad programming, but it is easier to write code that we understand and then explain. Especially at the learning stage.

Another major change incorporated is that the transaction number is added along with the transaction hash. This is the value of the variable tno, the top most for loop variable. This means that the first transaction, the Coinbase transaction's inputs will all have a value of 0, the second transaction's inputs will all have a value of 1 and so on and so forth. Thus, given a transaction hash, it is easy to acquire the transaction index number this output hash is part of, within the block. A change in the value of the tno variable signifies it is the inputs of a new transaction we are dealing with. There will be multiple inputs for every transaction, but all the inputs for the same transaction will have the same noofinputs for that transaction number.

While running through all the undo records, there is no filter set or any if statements that checks for a certain condition. The only check is on the empty undo blocks of size 1 as they are to be ignored

A revision of the undo block format. The first byte called first is the number of transactions in the blk block minus 1. The second byte is the number of inputs that are present in the first transaction counting starts from 0. Then comes the actual undo block data with the height, maybe version and ends with the amount and public key.

If there are three transactions in the block, trouble strikes? So far, we handled blocks with 2 transactions only. The above explanation holds true for the first undo transaction and the second transaction in the block, but not for more than 2. There is an extra number added to the undo data per transaction that must be read for 3 or more transactions. Let's look at the significance of this number.

There is an if statement to handle this special case. The third member of the inputlist list is the transaction number tno, which is the physical index of a transaction in a block. The value of this variable starts from 0 and goes on to one less than the number of transactions in the block.

Our code checks if the third member or variable tno matches the member in the previous list. If it is the same, then the transaction is part of the same transaction hash. The members of the list are accessed using undoindex and undoindex - 1. The current one with the previous member of the same list. Let's assume that the previous value of the tno variable

is 1 and the current value is 2. In such a case, the current input is the first input of a different transaction hash, vis-a-vis the previous member whose input is now the last input of another transaction hash.

In the program, the `readint` function reads the value stored in the undo block. This value is stored in a variable, `noinputs1`. This value must match with what the `inputlist` list returns. The third member in `inputlist` is a transaction number, the fourth is the number of inputs. Thus, the value read now from the undo data is number of inputs. This check is carried out after the second transaction only.

This program sort of matches the time taken to run the last UTXO program. The number of blocks may be small, as in half a million, but some of them have 1000s of transactions.

There is one last hash type to be explained and that is `multisig`. An error occurs when this hash type is left out. When the error popped up the first time, the first byte was `0x80` so we had to move the offset variable by 1. The error is seen again a few days later despite the first byte being `0x81`, so we fixed it again. However, this trend continued. So, in frustration, we used the `>=` sign and not `==`. Now there are no errors. Not a pretty sight showing this permutation and combinations in code, we are going blind.

This program could take weeks to run. This is not a typo, it takes that long. So, it is recommended that you copy this program into say 5 programs. Then do a hundred thousand blocks in each program by changing the range start and end points in the main for statement. Or better still, use a spare laptop. Save all the code in Dropbox folder and have a dedicated laptop only to run the programs uninterrupted for weeks on end.

```
bitcoin-cli getrawtransaction fbde5d03b027d2b9ba4cf5d4fecab9a99864df2637b25ea4cbcb1796ff6550ca 1
error code: -5
error message:
No information available about transaction
```

Run the above command on a machine where transaction index has not been enabled. In this case, both our outputs have been spent and hence they do not figure in the UTXO data set.

```
bitcoin-cli getrawtransaction fbde5d03b027d2b9ba4cf5d4fecab9a99864df2637b25ea4cbcb1796ff6550ca 1
{
  "txid": "fbde5d03b027d2b9ba4cf5d4fecab9a99864df2637b25ea4cbcb1796ff6550ca",
  "size": 258,
  "version": 1,
```

Now we are on a machine that has the transaction index enabled; the displayed output is truncated. Do observe that the transaction is in block 100001 and both outputs of this transaction have been spent.

CHAPTER 27

peers.dat and banlist.dat

In this chapter, we will read two small .dat files. The first is the smallest banlist.dat and the second is peers.dat. The file extensions may be .dat but it has nothing to do with the Berkley DB database.

Copy this file, banlist.dat on to your current folder. The file size is only 37 bytes. We like chapters like this one where our output matches yours.

Reading the Magic Number of the banlist.dat File

```
ch2701.py
f = open("./banlist.dat", "rb")
ban = f.read()
print "%x%x%x%x" % (ord(ban[0]), ord(ban[1]), ord(ban[2]), ord(ban[3]))
print "%x" % ord(ban[4])
```

Output
f9beb4d9
0

The file is read into memory and the first 4 bytes are displayed. They are the magic number, f9beb4d9. A few things like a magic number do not change at all.

The next byte is 0 which indicates that there is no IP address or no node is banned. Do not be surprised if you get a number larger than 1.

Checking the Hash of the banlist.dat File

```
ch2702.py
from cfuncs import *
f = open("./banlist.dat", "rb")
bandata = f.read(5)
hashban = f.read(32)
print hashban.encode('hex')
hash = chash(bandata)
print hash.encode('hex')
```

Output
7acbd11485c1d69bbadfe4faae871bbab4b0960dcd5a7dfaf0404e91f38635d0
7acbd11485c1d69bbadfe4faae871bbab4b0960dcd5a7dfaf0404e91f38635d0

The structure of this file format looks like the rev file. The first five bytes of the file, banlist.dat are read in a variable,

bandata. The first 4 bytes are for the magic number and the next byte is 0. There are no IP addresses in this file. This byte is not a size but a count of the IP addresses.

The next or the last 32 bytes of the file is a Sha256 double hash value. A hash value is computed of the first 5 bytes, i.e. the banlist data only and it is stored in a variable, hash. The hash value stored on file is read into a variable called hashban. This computed value in hash matches the hash value in the hashban variable. This confirms that the banlist.dat file is not corrupted aka the rev/undo file. This program only works because currently there are no IP addresses stored in the file, banlist.dat.

The banlist.dat file has no data by default.

```
bitcoin-cli listbanned
[
]
```

The listbanned command suggests the same thing.

The setban command adds an IP address to the banlist.dat file.

```
bitcoin-cli setban "10.0.0.1" "add" 12
bitcoin-cli listbanned
[
  {
    "address": "10.0.0.1/32",
    "banned_until": 1471537025,
    "ban_created": 1471537013,
    "ban_reason": "manually added"
  }
]
```

The output confirms that one IP address has been added.

Checking the Hash Value with One IP Address

```
ch2703.py
from cfuncs import *
f = open("./banlist.dat", "rb")
bandata = f.read(59)
hashban = f.read(32)
print hashban.encode('hex')
hash = chash(bandata)
print hash.encode('hex')
```

Output

```
c2a989eb3c36cdd0b08d79c00b2b8e8cc4e4e3cd0b050212fc198faa3bf1eb12
c2a989eb3c36cdd0b08d79c00b2b8e8cc4e4e3cd0b050212fc198faa3bf1eb12
```

Running the same program in the year 2017

Output

```
ef8ea284602c848852719e28e79d6cc7dc481cc1b91d3c49abbd377c6317b71c
ef8ea284602c848852719e28e79d6cc7dc481cc1b91d3c49abbd377c6317b71c
```

On adding one IP address to the file, the file size increases to 91 bytes. We have already accounted for the magic number 4 bytes, 1 byte for the number of IP addresses and 32 bytes for the final hash value. Thus, we are not wrong to state that each IP address takes up 54 bytes ($91 - 32 - 5$). This program reads the first $54 + 5$, 59 bytes and then computes its hash value. The hash value matches the value stored on disk. The length is hard coded just to confirm the right size.

We forgot to mention that the above program will only work if you copy the file banlist.dat to the current folder from the Bitcoin folder. The IP address is added in the file banlist.dat stored in the Bitcoins folder only.

Add One More IP Address to the Banned List.

```
bitcoin-cli setban "10.0.0.2" "add" 11
bitcoin-cli listbanned
[
  {
    "address": "10.0.0.2/32",
    "banned_until": 1471539851,
    "ban_created": 1471539840,
    "ban_reason": "manually added"
  }
]
```

We are a little upset that we still have only one address in the file. The reason here is that the number 11 or 12 is the duration in seconds, the address would stay banned. Quickly add two banned addresses and save the banlist.dat file on current folder.

```
bitcoin-cli setban "10.0.0.1" "add" 12
bitcoin-cli setban "10.0.0.2" "add" 12
bitcoin-cli listbanned
[
  {
    "address": "10.0.0.1/32",
    "banned_until": 1471539992,
    "ban_created": 1471539980,
    "ban_reason": "manually added"
  },
  {
    "address": "10.0.0.2/32",
    "banned_until": 1471540000,
    "ban_created": 1471539988,
    "ban_reason": "manually added"
  }
]
```

The size of the file banlist.dat is now 145 bytes. Let's do a rough calculation manually, 5 initial bytes and then the 32 bytes' hash, so the total is 37 bytes. The remaining 108 bytes when divided by 54 (the size of each IP address) gives a value of 2, so there are 2 IP addresses.

Understanding the Output of the Listbanned Command

```
ch2704.py
import time
print "banned_until 1471539992 %s" % time.ctime(1471539992)
print "banned_created %s" % time.ctime(1471539980)
print "banned_until 1471540000 %s" % time.ctime(1471540000)
print "banned_created 1471539988 %s" % time.ctime(1471539988)
```

Output

```
banned_until 1471539992 Thu Aug 18 22:36:32 2016
banned_created Thu Aug 18 22:36:20 2016
banned_until 1471540000 Thu Aug 18 22:36:40 2016
banned_created 1471539988 Thu Aug 18 22:36:28 2016
```

Let's now understand the two fields, banned_util and ban_created, with a simple program. The first banned IP address was created on Thursday the 18th of August at 10:36 at night past 20 seconds. This value is assigned to the banned_created field. Since the last parameter was 12, the ban lasted for only 12 seconds till 10:36 past 32 seconds. This value is shown in the field banned_until. The second ban was created at 22:36:28 and lasted till 22:36:40. It is easy understanding one concept at a time. This is very old data.

Let's us start with a clean slate by running the command, clearbanned.

```
bitcoin-cli clearbanned
bitcoin-cli listbanned
[
]
```

Now add the following IP address to our banned list.

```
bitcoin-cli setban "1.6.7.8/0" "add" 1000
bitcoin-cli setban "2.6.7.8/8" "add" 1000
bitcoin-cli setban "3.6.7.8/16" "add" 1000
bitcoin-cli setban "4.6.7.8/24" "add" 1000
bitcoin-cli setban "5.6.7.8/32" "add" 1000
```

The ban_reason indicates that the bitcoin-cli command added the banned IP address, manually. Adding one more banned address displays two different outputs. The reason being, the data and time is stored in the file bannlist.dat along with the IP address. The time is different even though the IP address are the same. When data changes, the hash changes.

```
bitcoin-cli listbanned
[
  {
    "address": "0.0.0.0/0",
    "banned_until": 1471547487,
    "ban_created": 1471546487,
    "ban_reason": "manually added"
  },
]
```

```

{
  "address": "2.0.0.0/8",
  "banned_until": 1471547488,
  "ban_created": 1471546488,
  "ban_reason": "manually added"
},
{
  "address": "3.6.0.0/16",
  "banned_until": 1471547488,
  "ban_created": 1471546488,
  "ban_reason": "manually added"
},
{
  "address": "4.6.7.0/24",
  "banned_until": 1471547488,
  "ban_created": 1471546488,
  "ban_reason": "manually added"
},
{
  "address": "5.6.7.8/32",
  "banned_until": 1471547488,
  "ban_created": 1471546488,
  "ban_reason": "manually added"
}
]

```

There are five IP addresses in our banned list.

Decoding Multiple IP Address Stored in the banlist.dat File

```

ch2705.py
from cfuncs import *
import time
f = open("./banlist.dat" , "rb")
fivebytes = f.read(5)
noip = ord(fivebytes[4])
print "Number of IP Addresses %d" % noip
for i in range ( 0 , noip):
    ip4 = f.read(12)
    print "ip6 unused %s" % ip4.encode('hex')
    ipaddress = f.read(4)
    print "IPAddress %s" % ipaddress.encode('hex')
    netmask = f.read(16)
    print "Network mask %s" % netmask.encode('hex')
    valid = f.read(1)
    print "Is IP address valid %d" % ord(valid)
    version = rint(f)

```

```
print "Version %d" % version
timec = rint(f)
print "Creation Time %s" % time.ctime(timec)
et1 = rint(f)
print "extra time %d" % et1
timeb = rint(f)
print "Banned Until %s" % time.ctime(timeb)
et2 = rint(f)
print "extra time %d" % et2
reason = f.read(1)
print "Reason of creation %d" % ord(reason)
print
```

Output

```
Number of IP Addresses 5
ip6 unused 00000000000000000000ffff
IPAddress 00000000
Network mask ffffffffffffffffffff00000000
Is IP address valid 1
Version 1
Creation Time Fri Aug 19 00:24:47 2016
extra time 0
Banned Until Fri Aug 19 00:41:27 2016
extra time 0
Reason of creation 2

ip6 unused 00000000000000000000ffff
IPAddress 02000000
Network mask ffffffffffffffffffff000000
Is IP address valid 1
Version 1
Creation Time Fri Aug 19 00:24:48 2016
extra time 0
Banned Until Fri Aug 19 00:41:28 2016
extra time 0
Reason of creation 2

ip6 unused 00000000000000000000ffff
IPAddress 03060000
Network mask ffffffffffffffffffff0000
Is IP address valid 1
Version 1
Creation Time Fri Aug 19 00:24:48 2016
extra time 0
Banned Until Fri Aug 19 00:41:28 2016
extra time 0
Reason of creation 2

ip6 unused 00000000000000000000ffff
```

```

IPAddress 04060700
Network mask ffffffff00
Is IP address valid 1
Version 1
Creation Time Fri Aug 19 00:24:48 2016
extra time 0
Banned Until Fri Aug 19 00:41:28 2016
extra time 0
Reason of creation 2

ip6 unused 00000000000000000000ffff
IPAddress 05060708
Network mask ffffffff
Is IP address valid 1
Version 1
Creation Time Fri Aug 19 00:24:48 2016
extra time 0
Banned Until Fri Aug 19 00:41:28 2016
extra time 0
Reason of creation 2

```

As a matter of abundant caution, always copy the banlist.dat file to the folder you are working in.

The initial 5 bytes are read in the variable fivebytes and the value in fifth byte is displayed. The output shows a value of 5, thus indicating there are 5 IP addresses. Each IP address takes up 54 bytes so the loop statement in the program loops 5 times, reading 54 bytes at a time. The variable noip decides on the iterations.

Let's understand the structure of the 54 byte IP address.

The first field is 16 bytes large. This is because a IPv6 address is 128 bits long whereas a IPv4 address is only 32 bits or 4 bytes large. But in today's day and age, all networking addresses are 16 bytes large where only the last 4 bytes are used. In our code, first, only 12 bytes are read and printed. Then, the last 4 bytes representing the IP address are read.

The network mask is either very easy or very difficult to understand. Whether we place /32 or not to the IP address, the default network mask is 32. The network mask is displayed with all FF's, so all bits are on.

Going backwards. The IP address starting with 4 ends with a /24. Therefore, consider only the last 24 bits of the IP address and hence the IP address displayed is 04060700. You can observe that the lower 8 bits or last 8 bits are zeroed out. This is the 8. It is like bitwise adding with 24 bits having 1 and the lower 8 bits having 0. The network address has the first 8 bits as 0's. There is a bitwise anding of the IP address and the network mask.

The IP address is displayed in this manner.

In the IP address 3.6.7.8/16, we are removing the first 16 bits so the IP address displayed is 3.6.0.0 and the network mask is ff0000. With /24, the first 3 bytes are zeroed out and the network mask is ff000000. The IP address is now 02.00.00.00. Finally, with /0, all the bits of the IP address are zeroed out. This delves further into details like sub-netting or a host and network address.

The next byte validates the IP address. All our addresses are valid. Then comes a 4-byte version field. We have seen the version field having 1 all the time. Lucky us !!!

As the Bitcoin code is written in C, the time is represented by 8 and not 4 bytes. In Python, only the first 4 bytes are read as an integer and then displayed. The time in English is obtained using the ctime function. The next 4 bytes of the

time are all 0's and hence ignored. The time is the creation time of the banned address in seconds. The next 8 bytes are the time at which the banned address will be valid again. Same treatment for time always. Ignore the second 4 bytes.

The last byte is the reason byte. Why was this IP address banned?

A value of 2 indicates that the IP address is manually added.

A value of 0 means that Bitcoin has no reason why this IP address was added.

A value of 1 means that the IP address was misbehaving by the bitcoin code and hence added. We have no idea who is to decide when IP addresses misbehave.

The source code lists only 3 possible values.

Let's move our attention to the file peers.dat.

Reading the Initial Fields of the peers.dat File

```
ch2706.py
f = open("./peers.dat")
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

magic = f.read(4)
nVersion = f.read(1)
nKeySize = ord(f.read(1))
key = f.read(nKeySize)
print "Magic %s" % magic.encode('hex')
print "Version %d" % ord(nVersion)
print "Key Size %d" % nKeySize
print "Key is %s" % reversehash(key.encode('hex'))
```

Output

```
Magic f9beb4d9
Version 1
Key Size 32
Key is 33a8c0a05a64bf626a1fe56205ab3770832e2a5b0c372f3bf2802e1b0d976c65
```

As before, copy the file peers.dat to the current folder. This file is about 212KB or 3.9 MB large. This implies that the file stores the IP addresses of many nodes/peers/ miners and more.

This file starts with the same magic number. The next byte is for the version, which for a long time will remain as 1. The next byte stands for the key size and the value is 32. We will understand shortly why this size must to be stored in the header.

The key is just a fancy word for a password. In this case, the password is 256 bits large or 32 bytes. But technically it can change over time. The Bitcoin ecosystem can be easily compromised if it is connected to certain malicious peers or malicious nodes. The Bitcoin source code explicitly lives on the trusts that the blockchain it has downloaded has not been tampered.

What is required here is a Stochastically IP address manager, not our words but taken from the source code comments. According to Google, a synonym for Stochastically is unpredictable.

Choosing the right node to download the blockchain is utmost crucial. Bitcoin chooses nodes in a very unpredictable

way. Therefore, the file peers.dat is very important. No one should decide the nodes one should connect to, to download the blockchain data.

The hash is reversed because in C++ code, it appears in the reverse form.

Reading 3 More Fields from the File peers.dat

```
ch2707.py
from cfuncs import *
f = open("./peers.dat")
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

magic = f.read(4)
nVersion = f.read(1)
nKeySize = ord(f.read(1))
key = f.read(nKeySize)
nNew = rint(f)
print "nNew %d" % nNew
nTried = rint(f)
print "nTried %d" % nTried
nUBuckets = rint(f) ^ (1 << 30)
print "nUBuckets %d" % nUBuckets
```

Output

```
nNew 1380
nTried 2285
nUBuckets 1024
```

After the key, there are three integers stored back to back. The numbers are too large for comfort. So, all that we do, is simply stop the bitcoin server, delete the peers.dat file and then rerun the server. This is what the log shows us.

```
2016-08-19 06:57:28 ERROR: Read: Failed to open file
/Users/vijaymukhi/Library/Application Support/Bitcoin/peers.dat
2016-08-19 06:57:28 Invalid or missing peers.dat; recreating
2016-08-19 06:57:28 Loaded 0 addresses from peers.dat 0ms
```

Some files can be deleted at will, like peers.dat. Bitcoin does not complain, it re-creates a new peers.dat file that is only 13k in size. Running the above program has a very different output.

Output

```
nNew 131
nTried 4
nUBuckets 1024
```

When we shut down the bitcoin server, the peers.dat file does not get saved to disk immediately, it gets written every 15 minutes. These three new variables and their use in for statements are explained in the next program. These numbers keep growing.

Deciphering the Entire File peers.dat

```
ch2708.py
from cfuncs import *
import time
import subprocess
import json
def rshort(f):
    return struct.unpack('>H', f.read(2))[0]
f = open("./peers.dat")
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
ips = []
ipsmain = ['37.34.48.17', '8.8.8.8', '192.3.11.20', '104.28.18.99', '176.9.45.239',
'192.3.11.24']
ipsnew = []
magic = f.read(4)
nVersion = f.read(1)
nKeySize = ord(f.read(1))
key = f.read(nKeySize)
nNew = rint(f)
print "nNew %d" % nNew
nTried = rint(f)
print "nTried %d" % nTried
nUBuckets = rint(f)
print "nUBuckets %08x" % nUBuckets
print "-----Start"
for i in range ( 0 , nNew):
    print "nNew i %d" % i
    version = rint(f)
    print "Version %d" % (version )
    time1 = rint(f)
    print "Time %08x %s" % (time1 , time.ctime(time1))
    nServices = r8bytes(f)
    print "nServices %d" % nServices
    ip6 = f.read(12)
    print "ipv6 %s" % ip6.encode('hex')
    ip4 = f.read(4)
    print "ipv4 %d.%d.%d.%d" % (ord(ip4[0]), ord(ip4[1]), ord(ip4[2]),
ord(ip4[3]))
    port = rshort(f)
    print "Port Number %d" % port
    ip6a = f.read(12)
    print "Source ipv6a %s" % ip6a.encode('hex')
    ip4a = f.read(4)
```

```

sipaddr = "%d.%d.%d.%d" % (ord(ip4a[0]), ord(ip4a[1]), ord(ip4a[2]),
ord(ip4a[3]))
if sipaddr not in ips:
ips.append(sipaddr)
print "Source ipv4a %s" % sipaddr
nLastSuccess = r8bytes(f)
print "nLastSuccess %s" % time.ctime(nLastSuccess)
nAttempts = rint(f)
print "nAttempts %d" % nAttempts
print
for i in range(0, nTried):
print "nTried %d" % i
version = rint(f)
print "Version %d" % (version)
time1 = rint(f)
print "Time %d %s" % (time1, time.ctime(time1))
nServices = r8bytes(f)
print "nServices %d" % nServices
ip6 = f.read(12)
print "ipv6 %s" % ip6.encode('hex')
newaddr = "%d.%d.%d.%d" % (ord(ip4[0]), ord(ip4[1]), ord(ip4[2]), ord(ip4[3]))
ip4 = f.read(4)
if newaddr not in ipsnew:
ipsnew.append(newaddr + ":" + "%d" % port)
print "ipv4 %d.%d.%d.%d" % (ord(ip4[0]), ord(ip4[1]), ord(ip4[2]), ord(ip4[3]))
port = rshort(f)
print "Port Number %d" % port
ip6a = f.read(12)
print "Source ipv6a %s" % ip6a.encode('hex')
ip4a = f.read(4)
sipaddr = "%d.%d.%d.%d" % (ord(ip4a[0]), ord(ip4a[1]), ord(ip4a[2]), ord(ip4a[3]))
if sipaddr not in ips:
ips.append(sipaddr)
print "Source ipv4a %s" % sipaddr
nLastSuccess = r8bytes(f)
print "nLastSuccess %d %s" % (nLastSuccess, time.ctime(nLastSuccess))
nAttempts = rint(f)
print "nAttempts %d" % nAttempts
print
for i in range(0, 1024):
size = rint(f)
if size > 0:
print "\nBucket Number %d %d—" % (i, size),
for j in range(0, size):
size1 = rint(f)
if size1 > 0:

```

```
    print "%d:" % size1 ,
print f.tell()
print ips
for i in ips:
    if i in ipsmain:
        print "%s in DNS" % i
    else:
        print "%s NOT in DNS" % i

raw = subprocess.check_output(["bitcoin-cli" , "getpeerinfo"])
raw = json.loads(raw)
for i in raw:
    if i["addr"] in ipsnew:
        print "IP Address %s Found" % i["addr"]
    else:
        print "IP Address %s Not Found " % i["addr"]

print ipsnew
```

Output

```
nNew 125
nTried 9
nUBuckets 40000400
-----Start
nNew i 0
Version 120100
Time 57b37c91 Wed Aug 17 02:20:25 2016
nServices 1
ipv6 00000000000000000000ffff
ipv4 50.141.229.114
Port Number 8333
Source ipv6a 00000000000000000000ffff
Source ipv4a 37.34.48.17
nLastSuccess Thu Jan 1 05:30:00 1970
nAttempts 0

nNew i 1
Version 120100
Time 57b417ff Wed Aug 17 13:23:35 2016
nServices 1
ipv6 00000000000000000000ffff
ipv4 93.190.138.234
Port Number 8333
Source ipv6a 00000000000000000000ffff
Source ipv4a 37.34.48.17
nLastSuccess Thu Jan 1 05:30:00 1970
nAttempts 0

Bucket Number 1012 1— 8:
```

Bucket Number 1023 1— 43:

12958

['37.34.48.17', '8.8.8.8', '192.3.11.24', '104.28.19.99', '0.0.0.0', '176.9.45.239']

37.34.48.17 in DNS

8.8.8.8 in DNS

192.3.11.24 in DNS

104.28.19.99 NOT in DNS

0.0.0.0 NOT in DNS

176.9.45.239 in DNS

IP Address 37.18.74.232:8333 Found

IP Address 51.255.91.67:8333 Found

IP Address 91.121.211.8:8333 Found

IP Address 95.97.151.87:8333 Found

IP Address 71.170.89.148:8333 Found

IP Address 167.114.174.212:8333 Found

IP Address 5.135.183.24:8333 Found

IP Address 93.185.176.84:8333 Not Found

['84.92.52.187:8333', '51.255.91.67:8333', '173.224.240.38:8333',

'37.18.74.232:8333', '71.170.89.148:8333', '91.121.211.8:8333',

'167.114.174.212:8333', '5.135.183.24:8333', '95.97.151.87:8333']

The new output

Output

nNew 62108

nTried 333

nUBuckets 40000400

-----Start

nNew i 0

Version 130100

Time 587675b7 Wed Jan 11 23:43:11 2017

nServices 13

ipv6 00000000000000000000ffff

ipv4 80.223.134.5

Port Number 8333

Source ipv6a 00000000000000000000ffff

Source ipv4a 8.8.8.8

nLastSuccess Thu Jan 1 05:30:00 1970

nAttempts 0

nTried i 312

Version 130100

Time 1484271787 Fri Jan 13 07:13:07 2017

nServices 67108877

ipv6 00000000000000000000ffff

ipv4 190.85.201.37

Port Number 8333

Source ipv6a 00000000000000000000ffff

```
Source ipv4a 139.162.211.181
nLastSuccess 1483396257 Tue Jan 3 04:00:57 2017
nAttempts 5
```

This one very large program has too many concepts explained in it. Personally, we don't like large programs. Just for your information, we understood the file format of peers.dat by observing what the original source code did, as always. We added plenty of printf statements in the original source code. In this way, we tracked down the steps executed while writing to this file.

Coming back to the code, there is an empty list called ips. Our program adds some IP addresses in string format to this list after some time. The ipsmain list has 6 IP addresses in dotted decimal notation but as strings. When the bitcoin server starts, it connects to a Bitcoin node/miner/peer and downloads the blocks.

How does the Bitcoin server know which nodes or peers to connect to? It cannot choose IP addresses randomly. Plus, they also must be genuine ones. The answer lies in the Bitcoin source. There are six domain names hardcoded. So, we found the IP addresses associated with these domain names and added them to the ipsmain list.

The following are the domain names along with their IP addresses which are not present in the source.

```
bitcoin.sipa.be 37.34.48.17
bluematt.me 8.8.8.8
dashjr.org 192.3.11.20
bitcoinstats.com 104.28.18.99
bitcoin.jonasschnelli.ch 176.9.45.239
dashjr.org 192.3.11.24
```

The source code reveals that along with the domain names, there are the names of people like Dr. Pieter Wuille.

Dr. Wuille has written the code of the IP address manager that creates the peers.dat file. The best thing we like about Dr. Pieter is that despite contributing a lot of code to the Bitcoin source, he also has time to answers questions on Bitcoins. He is active on sites like stackexchange.com. Dr. Pieter Wuille is also the main author of Segregated witnesses. One of my other legends in the Bitcoin world is Mr. Gavin Andresen. The list goes on.

Let's continue further and look at the nNew, nTried and UBuckets variables.

There are a series of identical structures giving details about each node stored in a sequential order. There are two different lists now. The number of members of the first list is decided by the nNew field and the size of the second list (identical one) is decided by the nTried field.

The buckets are a concept unique to the IP address manager. Here, the IP addresses are placed in buckets. Addresses that are not tried or not yet verified or not used are placed in 1024 new buckets. From a single bucket, 64 random addresses can be chosen.

The tried buckets have IP addresses which are available for use and have been tried or used by these nodes earlier. The magic number 8 comes from the fact that to download the blockchain, a connection is made to 8 nodes.

This can be verified when you connect to the Bitcoin-Qt client, it shows a maximum of 8 connections being made.

In the program, the selected bucket uses cryptographic hashing (this key field is not explained) so that a local attacker (sitting on our machine) does not poison the nodes to coerce us into downloading blockchains from nodes it wants us to download from. The bitcoin source trusts that the blockchain copy on our local machine does not contain malicious data.

The number of buckets is fixed as a constant value of 1024 even though the output says otherwise. The IP address manager version 0 in the past skipped the version number in deserialization.

Therefore, the number of buckets 1024 is xored by a very large number 2^{30} so that the old address manager detects this value as incompatible. The number of buckets is kept as a constant value of 1024. This knowledge comes after reading the comments in the source code.

Back to code now. We first display the new nodes using a for statement. The value in the nNew variable decides on the number of nodes. The first field is obviously a version field of our bitcoin client, which is not upgraded from version 12.01. The next output shows the latest client version, 0.13.01. We are now running version 0.15 so we see version 15.9900.

The next 4 bytes and not 8 bytes are the network time, the time we deleted the peers.dat file. Either take the August 17th, 2016 time or the newer Jan 11, 2017 time or Tue Oct 10, 2017. This happens when you run code spread over versions.

Then there is an 8-byte services field which is always 1 or 13 or 15, today. This is followed by a 16 byte ipv6 field. As explained earlier, the first 12 bytes are ignored and only the last 4 bytes are taken. We connect to this IP address of the node to download the blockchain. Someday, we will download the blockchain data from these nodes. As we are also a valid or equal node, our IP address also must be present on someone's peers.dat file sitting in some Bitcoin folder somewhere in the galaxy.

The peers.dat file is a library or repository of nodes from where we can download the blockchain files.

For a connection, we need a port number; the Bitcoin universe only uses port 8333. The next field in line is for the port number and it gives the same value. A repetition if you recall. If you see another port number here, you are dreaming.

A node is very important from security point of view. The next 16 bytes have the IP address of the source of the current node.

The IPv4 addresses is converted into a dotted decimal notation string, a series of 4 numbers from 0 to 255 and then checked in the ips list. If it is not present, then the current IP address is added to the ips list. We use the variable ip4a for the ip address to be added to the list.

At the end of the program, there is a display of all unique IP addresses found in the nNew list.

Let's now figure out if these IP addresses are present in the master list of known IP addresses i.e. the list variable, ipsmain.

The field nLastSuccess gives information on the last successful connection made to the node. This 8-byte field is a time value. The time function displays this value. The last field value is the number of attempts made for connection. This process is used for every new node as these nodes are stored back to back. The same code is repeated in the for loop depending upon the value in the nTried variable.

The second output of the 2017 vintage shows five attempts were made to connect. Most or the successful times are dated 1970. Our output has a date of 2017. This output comes from the second for loop.

Now to display the buckets contents in the file peers.dat. The outer for loop has constant size of 1024 buckets to read. The first int figures out the number or size of buckets. First the number of buckets with valid indexes is displayed, followed by the index values.

The second for loop starts from 0, and loops for the number of index values per bucket. This is the size variable. The bucket size value is displayed and its index value if it is not 0. Too much output to display.

In the tried list, every unique IP address in ip4a which has had a connection is saved in the ipsnew list.

The command getpeerinfo is used next to obtain a list of all the peers or nodes that we are currently connected to. These 8 node details are stored back to back in a list with some zillion details about each node. The Bitcoin server does not have to be connected to 8 nodes all the time. This number is dynamic. Our Bitcoin server can connect to a maximum of 8 nodes.

The field `addr` is extracted and then compared with the IP address returned by the method `getpeerinfo` and the list variable `ipsnew`, which is from the file `peers.dat`. Out of 8, 7 IP addresses match. The IP address 93.185.176.84 does not match. The connection times are also the same with a minor difference of 1 second.

```
IP Address 37.18.74.232:8333 Found
IP Address 51.255.91.67:8333 Found
IP Address 91.121.211.8:8333 Found
IP Address 95.97.151.87:8333 Found
IP Address 71.170.89.148:8333 Found
IP Address 167.114.174.212:8333 Found
IP Address 5.135.183.24:8333 Found
IP Address 107.150.40.234:8333 Found
```

After a wait of 30 minutes, when the same program is executed, all 8 IP address used by Bitcoin connected nodes match. These IP addresses are also different from the earlier result as Bitcoin servers keep connecting to different nodes all the time. Most the IP addresses will say Found.

There is one for loop that we ignored. The `ips` list is scanned. It has the nodes or the IP addresses one can connect to. We check if the IP address is present in the `ipsmain` list, our hard-coded list of original IP addresses.

The final print command with `ipslist` shows our IP address along with the other IP address. Finally, it is part of the `peers.dat` file. The address will not be at the end of the file but 32 bytes from the end. The next set of bytes are for the final hash value.

We have not delved into details on the two sets of IP addresses, new and tried, created by Bitcoin. The meaning of the indexes stored in the 1024 buckets is still pending. As we said earlier, we ignore the key field. We tried adding a node manually using the command `addnode`, but the node never got added to `peers.dat`. The C++ source has all the answers to it but the code was too difficult to read. This chapter simply shows the effort Bitcoin takes to connect to a peer. We or our peer stands no chance of someone connecting to us.

Cross Checking the Hash Stored in the File `peers.dat`

```
ch2709.py
from cfuncs import *
f = open("./peers.dat")
first = f.read()
calchash = first[:-32]
diskhash = first[-32:]
hash1 = chash(calchash)
print hash1.encode('hex')
print diskhash.encode('hex')
```

Output

```
b43fb5e01cb1cd2a50c94e204c5d6a7ce211cfe91a6b83ce1516a4b50d7496ab
b43fb5e01cb1cd2a50c94e204c5d6a7ce211cfe91a6b83ce1516a4b50d7496ab
```

A 32-byte field is always assumed to be a double SHA hash value. The entire `peers.dat` file is read into a variable called `first`. Then the last 32 bytes of the file are extracted for the hash value. Python makes it very simple with the slice operator; the second parameter is the end parameter.

First `[:-32]` reads the initial bytes less 32 bytes of the file stream in first. Similarly, `first [-32:]` reads the last 32 bytes in the stream. Sounds logical, but after some soul searching. A hash value of the data collected in `calchash` is computed using the `chash` function. The hash value on the disk and the computed one match completely.

The technique used to calculate the hash value with the `peers.dat` file is already seen in the undo files as well as the checksum. Consistency thy name is Bitcoin?

CHAPTER 28

Miners, Blocks and More

The Bitcoin ecosystem survives only on miners. Without miners, it would be like a country without people. Therefore, an entire chapter devoted on miners, how they work, why does it cost a billion dollars to be a miner and so on and so forth. The next series of programs deal with what miners eat for breakfast, lunch and dinner, and it is blocks and block and more blocks.

Displaying the Chainwork Field and its Difference

```
ch2801.py
import subprocess
import json
raw = subprocess.check_output(["bitcoin-cli", "getblockhash", "46554"])
raw = subprocess.check_output(["bitcoin-cli", "getblock", raw])
raw = json.loads(raw)
nonce = raw["nonce"]
print "Nonce %d" % nonce
chainwork = raw["chainwork"]
chainwork = int(chainwork, 16)
print "Block 46554 Chainwork %d" % chainwork

raw = subprocess.check_output(["bitcoin-cli", "getblockhash", "46555"])
raw = subprocess.check_output(["bitcoin-cli", "getblock", raw])
raw = json.loads(raw)
chainwork1 = raw["chainwork"]
chainwork1 = int(chainwork1, 16)
print "Block 46555 Chainwork %d" % chainwork1

raw = subprocess.check_output(["bitcoin-cli", "getblockhash", "46556"])
raw = subprocess.check_output(["bitcoin-cli", "getblock", raw])
raw = json.loads(raw)
chainwork2 = raw["chainwork"]
chainwork2 = int(chainwork2, 16)
print "Block 46556 Chainwork %d" % chainwork2

print "diff1 %d" % (chainwork1 - chainwork)
print "diff2 %d" % (chainwork2 - chainwork1)
```

Output

```
Nonce 1388
Block 46554 Chainwork 284997795251337
```

```

Block 46555 Chainwork 285017402775924
Block 46556 Chainwork 285037010300511
diff1 19607524587
diff2 19607524587

```

The json data of block number 46554 is used in the program. This block had the lowest nonce, 1388. The field displayed is chainwork; it was not seen before in the Bitcoin blockchain. It stores a number, so the int function is used to convert the string into an actual integer.

The chainwork field of the next two blocks, block numbers 46555 and 46556 is also displayed. The difference between two consecutive chainwork values is calculated and surprisingly, the difference between them is the same, 19607524587. This value is much larger than the nonce value. Just a reminder, nonce has a count value, as in number of times the Sha256 hash has been recalculated. The problem comes in when the value of nonce rolls over and starts with 0. A 32-bit number in the Bitcoin world is a very small number.

The nonce is not the only field that changes periodically in the Bitcoin block header. The date time field also changes. Now to understand this number, 19607524587.

Adding the Chainwork for all Bitcoin Blocks

```

ch2802.py
import subprocess
import psycpg2
import json
import httpplib
import base64
conn = psycpg2.connect("dbname='postgres' user='postgres' host='localhost'
password='accord'")
cur = conn.cursor()
def getwork1(nCompact):
    a = pow(2 , 256)
    exp = nCompact >> 24
    mant = nCompact & 0xffffffff
    target = (mant * (1<<(8*(exp - 3))))
    target = target + 1
    b = a / target
    return b
totchainwork = 0
for blkno in range(0, 500000):
    s1 = "Select hash from blockhash where blkno = %d" % (blkno )
    cur.execute(s1)
    row = cur.fetchone()
    hash = row[0]
    raw = subprocess.check_output(["bitcoin-cli" , "getblock" , hash])
    raw = json.loads(raw)
    bits = raw["bits"]
    bits = int(bits , 16)
    work1 = getwork1(bits)

```

```
totchainwork = totchainwork + work1
chainwork = raw["chainwork"]
chainworkint = int(chainwork,16)
if totchainwork== chainworkint:
    print "Chainwork matches at blockno %d" % (blkno )
else:
    print "Chainwork does not match at blockno %d" % blkno
    print "Chainwork %s" % chainwork
    print "bits %x work %x " % (bits , work1 )
    exit()
```

Output

```
Chainwork matches at blockno 0 work1 4295032833 totchainwork 4295032833
Chainwork matches at blockno 60479 work1 71379488893 totchainwork 898263146672640
Chainwork matches at blockno 60480 work1 74654062325 totchainwork 898337800734965
```

This program starts from the genesis block and checks the chainwork field.

For some reason, the website blockchain.info also ignores the chainwork field. This field is not physically present in the blk files.

In one of the previous chapter, we learnt how miners used the bits/difficulty field, basically it was used to check if they have successfully completed the proof of work. The difficulty field verifies if the found hash value matches the condition, wherein it must have a value in a certain range.

Back to the code. Each block hash value is fetched from the blockhash table, but one at a time. The getblock command with the block hash is used to obtain the block data. Then the field called bits is extracted from this json converted block data. Further, the value in this string field is converted into an integer using the int function.

The getwork1 function is called thereafter and it is given the value of the bits field. In this function, the value in the bit field is converted into some value. The chainwork field uses this value. The maximum value a hash can represent is 2 to the power of 256. A 32-byte hash is 256 bits large. The variable a stores the largest value a hash can ever hold, that is 2 to the power of 256.

Then, using code from the difficulty chapter, the bits are broken up into the exp and mant variables. The target is computed in the same way and one is added to it. There is a simple relation between this newly calculated target value and the number of hashes theoretically needed and that is 2 raised to the power of 256.

Getting a hash that meets our condition is a random act or an act of God.

The chainwork field goes a step further. It has a number that represents the number of hashes computed to mine this block. The best chain value is the total value of all the individual chainworks per block and it will be the largest value of all the blocks in the blockchain. This field is also used to compare different chains.

The value received in the work1 variable is added to the totchainwork variable, which carries a running total.

The chainwork field is read as string and then converted into an int. The total hashes availed so far are compared with the values in the chainwork field. If we are on the right track, the numbers will match.

In short, the bits field indirectly gives a running total of the theoretical hashes that have been computed so far, on all blocks in the blockchain. The output shows that the value in the totchainwork variable obviously increases with every block as we are adding the value stored in the work1 variable. This running total must match the value of the chainwork field stored in the memory of the blockchain, not on disk.

In October 2017, no errors reported due to the addition of segregated witnesses.

For the first-time Python scores over C. In C/C++ code, the largest size is a long, so it is extremely painful and tedious to compute this chainblock value. Python treat it as a joke.

```
create table minerip (blkno integer, hash varchar(64) , timeinblock bigint,
timeactual bigint , ipadderss varchar(128))
```

Adding Blocks into a Table to Understand the Time Field in the Block Header

```
ch2803.py
import socket
from cfuncs import *
import datetime
import sys
import subprocess
import json
import time
import psycpg2
conn = psycpg2.connect("dbname='postgres' user='postgres' host='localhost'
password='accord'")
cur = conn.cursor()

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def sendversionpacket(sock):
    magicbytes = "f9beb4d9"
    command = "version\x00\x00\x00\x00\x00"
    commanddecoded = command.encode('hex')

    str1 = '62ea' + '0' * 166
    checksumhash1 = chash(str1.decode('hex'))
    checksumhash1 = checksumhash1[:4]

    packetlength = len(str1)/2
    packetlengthstr = "%08x" % packetlength
    str = magicbytes + commanddecoded + reversehash(packetlengthstr) +
checksumhash1.encode('hex')
    str2 = str + str1
    str2 = str2.decode('hex')
    sock.sendall(str2)

def decodeversion(payloadbytes):
    (version , services ) = struct.unpack("l8s" , payloadbytes[: 12])
    (timestamp , addr_recv , addr_from) = struct.unpack("Q26s26s" ,
payloadbytes[12:72])
    st = datetime.datetime.fromtimestamp(timestamp).strftime('%d-%B-%Y
%H:%M:%S')
    (recvservices , ipaddress , ipv4address , port1 , port2) =
struct.unpack("8s12s4scc" , addr_recv)
```

```
portno = ord(port1) * 256 + ord(port2)
(fromservices , fromipaddress , fromipv4address , fromport1 , fromport2) =
struct.unpack("8s12s4scc" , addr_from)
fromportno = ord(fromport1) * 256 + ord(fromport2)
(nonce , lengthstr ) = struct.unpack("Qc" , payloadbytes[72:81])
noncestr = "%x" % nonce
lengthstr = ord(lengthstr)
unpackstr = "%ds" % lengthstr
(useragent ) = struct.unpack(unpackstr , payloadbytes[81:81+lengthstr])
newlength = 81+lengthstr
(blockheight , relay ) = struct.unpack("lc" , payloadbytes[newlength:newlength+
4 + 1])

magicbytes = "f9beb4d9"
command = "verack\x00\x00\x00\x00\x00\x00"
commanddecoded = command.encode('hex')
length = "%08x" % 0
checksum = chash(payloadbytes)
checksumhash = checksum[:4]
checksumhash = checksumhash.encode('hex')
str = magicbytes + commanddecoded + length + checksumhash
str = str.decode('hex')
sock.sendall(str)
#print "VerAck Send"

def decodeverack(payloadbytes):
    #print "Decoding VerAck Packet"
    pass

def decoding(payloadbytes):
    #print "Decoding Ping Packet"
    if len(payloadbytes) == 8:
        (nonce ,) = struct.unpack("Q" , payloadbytes[:8])
        #print "Nonce is %x" % nonce
        nonce = "%x" % nonce
    else:
        nonce = "0000"
        nonce = reverseahash(nonce)
        checksum = chash(nonce.encode('hex'))
        checksumhash = checksum[:4]
        checksumhash = checksumhash.encode('hex')
        magicbytes = "f9beb4d9"
        command = "pong\x00\x00\x00\x00\x00\x00\x00\x00"
        commanddecoded = command.encode('hex')
        length = "%08x" % 8
        lengthrev = reverseahash(length)
        noncelen = len(nonce)
        diff = 16 - noncelen
```

```

    str = magicbytes + commanddecoded + lengthrev + checksumhash + nonce + '0' *
    diff
    str = str.decode('hex')
    sock.sendall(str)

dictionary = {0:"Error" , 1:"Transaction" , 2:"Block" , 3:"Filtered Block"}
def decodeinv(payloadbytes, length , sock):
    getdatversionbytes = payloadbytes
    (count , ) = struct.unpack("c" , payloadbytes[:1])
    count = ord(count)
    payloadbytes = payloadbytes[1:]
    if count == 253:
        (count, ) = struct.unpack("h" , payloadbytes[ : 2])
        payloadbytes = payloadbytes[2:]
    for i in range ( 0 , count ):
        (type , hash , ) = struct.unpack("l32s" , payloadbytes[i * 36 : i*36 + 36])
        if type == 2:
            print "Hash Type is %s" % dictionary[type]
            print "%s" % reverseahash(hash.encode('hex'))
            hash = reverseahash(hash.encode('hex'))
            actualtime = time.time()
            time.sleep(60)
            raw = subprocess.check_output(["bitcoin-cli" , "getblock" , hash])
            raw = json.loads(raw)
            blkno = raw["height"]
            timeinblock = raw["time"]
            ipaddress = ""
            s1 = "insert into minerip1 values(%d , '%s' , %d , %d , '%s') " % (blkno , hash ,
            timeinblock , actualtime , ipaddress )
            print s1
            cur.execute(s1)
            conn.commit()
            magicbytes = "f9beb4d9"
            command = "getdata\x00\x00\x00\x00\x00"
            commanddecoded = command.encode('hex')
            length = "%08x" % length
            lengthrev = reverseahash(length)
            checksum = chash(getdatversionbytes)
            checksumhash = checksum[:4]
            checksumhash = checksumhash.encode('hex')
            str = magicbytes + commanddecoded + lengthrev + checksumhash +
            getdatversionbytes.encode('hex')
            str = str.decode('hex')
            sock.sendall(str)

def handlecommand(command , length , checksum , payload , sock):
    checksumhash = chash(payload)

```

```
checksumhashfirstfour = checksumhash[:4]
if checksum != checksumhashfirstfour.encode('hex'):
    pass
else:
    if "version" in command:
        decodeversion(payload)
    elif "verack" in command:
        decodeverack(payload)
    elif "ping" in command:
        decodeping(payload)
    elif "inv" in command:
        decodeinv(payload , length , sock)
    else:
        pass

sock = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
sock.connect(("bitcoin.sipa.be" , 8333))
sendversionpacket(sock)
while ( True ):
    bytesrecieved = sock.recv(0x10000)
    try:
        (magicnumber , command, payloadlength , checksum ) =
        struct.unpack("I12sII" , bytesrecieved[:24])
    except:
        pass
    checksumstr = "%08x" % checksum
    payload = bytesrecieved[24:24 + payloadlength]
    handlecommand(command , payloadlength , reversehash(checksumstr) ,
    payload , sock)
```

Output

Hash Type is Block

00000000000000000000be86422482d43413ec385f825c8f3ac4833670b8c7a2ce

insert into minerip values(489826 ,

'00000000000000000000be86422482d43413ec385f825c8f3ac4833670b8c7a2ce' ,

1508007012 , 1508007037 , '')

select * from minerip;

blkno	hash	timeinblock	timeactual
-------	------	-------------	------------

ipadderss			
-----------	--	--	--

489819			
--------	--	--	--

00000000000000000000e0c8769e0d8cfd1e04c5cd7197746881175dc3e92031a3			
--	--	--	--

1508004168	1508004174		
------------	------------	--	--

489820			
--------	--	--	--

000000000000000000007c37654966bab779d7f81f701c40a008d572796a1b32fc			
--	--	--	--

1508004297	1508004301		
------------	------------	--	--

(8 rows)

We achieve two things with this one program.

There is a field called time in the 80-byte block header. What time does this field represent? The standard answer will be Block creation time.

Let's look at the Bitcoin wire protocol.

The program from the Networking chapter is used here where it handles a block with an inventory or inv message. The inv message is sent when a new transaction or a block makes a debut on the Bitcoin network. Networking code is always flaky so please have patience. And a block is never seen every day, just once every 10 minutes on an average. The decodeinv function gets called when an inv message comes in. The type field with a value of 2 indicates that it is a new block.

First, the hash type is displayed in a string format, which is Block and then the block hash value. This hash value must be reversed before it can be used. The python function, time determines the current date and time of receiving the block message on our computer.

There is a pause/sleep for a minute. The reason being, the Bitcoin server receives the block inv message at the same time when our program receives the same block message from the same or another peer. The Bitcoin server takes some time to process this message. Our program must respond to the block command after the server processes the message. An exception is thrown if there is no wait time. There can be multiple clients on the same machine but there will be only one server (bitcoind) on one machine. The actual time of arrival of the block is saved before sleeping.

The bitcoin-cli program is given the newly created block hash value and the getblock method returns all the block details. Our interest is only in the time field and the block number or height field. The two fields, the time we received the block, actualtime and the time written in the block header, timeinblock are added into the minerp table along with other details like the block number.

We wanted an output like the blockchain.info website, i.e. display the IP address of the miner that mined the block. Therefore, the table minerip has a field called ipaddress. After some trial and error, we realized that a connection is never made to a miner who mined the block, but only to a random peer. There are many peers out there who do only one job. They send blocks to people who will never ever be miners, me included. The blockchain.info discerns the IP address that sent out the message 'I have a block successfully mined'. This is the IP address of the peer who sent the website a block inv message and not the miner who actually mined the block. As we will see later, there is another way of finding out the miner's name.

On paper at the very least, there are only two computers or peers that can ever exist on a Bitcoin network. One is me or the miner and the second is the computer it connects to. It is one-to-one connection. There can be many computers on the Bitcoin network but in a peer-to-peer connection, it is miner to computer. It is nowhere close to Facebook or other apps on social media where a connection is made to all other friends.

In the case of a web wallet, no one must store or know any IP address. Once again, when our own home-grown transaction is broadcasted, other than the peer it is sent to, no one should know the source IP address. When that peer broadcasts this transaction, it does not inform the Bitcoin world that Vijay Mukhi or this IP address created that transaction.

The blockchain may be very smart but it does not store information of any IP addresses. You will not find a single IP address stored in the blockchain. Thus, blockchain.info only knows which computer first broadcasted a certain IP address, it may/may not be the one that created that transaction or block. We will never know.

The blockchain.info therefore, never confirms the miner's address, they say relayed by; may be the miner, may not be the miner.

Once again like with all networking code, this program may not work the first time. It will hang till the cows come home. We are sure the same will happen with Bitcoin Core 0.16.

Making Sense of the Time Stored in a Block Header and When We Receive the Block

```
ch2804.py
import psycopg2
import time
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost'
password='accord'")
cur = conn.cursor()
for blkno in range (489819 , 489822):
    s1 = "Select * from minerip where blkno = %d" % (blkno )
    cur.execute(s1)
    row = cur.fetchone()
    if row is None:
        continue
    timeinblock = row[2]
    actualtime = row[3]
    print "(%d)Block %s Actual %s Diff %d" % (blkno , time.ctime(timeinblock) ,
    time.ctime(actualtime) , actualtime - timeinblock )
```

Output

```
(489819)Block Sat Oct 14 23:32:48 2017 Actual Sat Oct 14 23:32:54 2017 Diff 6
(489820)Block Sat Oct 14 23:34:57 2017 Actual Sat Oct 14 23:35:01 2017 Diff 4
(489821)Block Sat Oct 14 23:39:03 2017 Actual Sat Oct 14 23:39:15 2017 Diff 12
```

We were only interested in saving half a dozen blocks in our minerip table. These blocks start from 489819. Your mileage will definitely vary. We ran the program multiple times over a period, to get blocks randomly spread all over the day.

We see that the time we received the blocks matches with the time stored in the block header. The difference is less than 30 seconds which can be caused by network latency.

Also, the time we received the block is larger than the time in the block because the block will take some time in reaching us after it has been mined by a miner. This time lag due to the network is as low as 2 seconds for some blocks and up to 27 seconds for other blocks.

The Bitcoin wiki makes it very clear, that the block times are accurate only for an hour. This block time also changes the block hash value. It keeps changing during the interval of 10 minutes taken by the miners to mine the blocks. This only proves that it is not just the nonce that changes when the block hash value is computed.

The Bitcoin wiki adds another twist to this field time. The Wiki defines the time in the block header as when the miner started hashing the header. This makes sense. The wiki then adds a caveat: that this time is decided by the miner who puts in a time when it has finally calculated the hash as per the difficulty set.

If the miner calculates the hash rightly and then changes the time, the hash will also change randomly that it may not meet the difficulty condition.

There is a condition placed on this time field. It must be greater than or equal to the median time of the last 11 blocks. Also, a node will not accept a block where the time is more than 2 hours in the future as per their clock.

A Field Called Mediantime

```

ch2805.py
import psycpg2
import subprocess
import json
conn = psycpg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
index = 0
mtime = []
errors = 0
for blkno in range(0, 500000):
    s1 = "Select hash from blockhash where blkno = %d" % (blkno )
    cur.execute(s1)
    row = cur.fetchone()
    hash = row[0]
    raw = subprocess.check_output(["bitcoin-cli" , "getblock" , hash])
    raw = json.loads(raw)
    timeh = raw["time"]
    timem = raw["mediantime"]
    if blkno <= 10:
        mtime.append(timeh)
    else:
        mtime = mtime[1:]
        mtime.append(timeh)
    if blkno >= 11:
        #print mtime
        if timem != mtime[5]:
            errors = errors + 1
        T = [mtime[i] for i in [0 , 1 , 2, 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10]]
        if timem in T:
            print "(%d) timeheader %d timemedian %d Passes Errors %d" % (blkno ,
            timeh , timem , errors )
            pass
        else:
            errors = errors + 1
            print mtime
            print "(%d) timeheader %d timemedian %d difference %d Fails %d" % (blkno ,
            timeh , timem , timem - mtime[0] , errors )
    exit()

```

Output

```
(453891) timeheader 1487586756 timemedian 1487586275 Passes Errors 30679
```

One more field that needs attention is the mediantime field. Time to dust up our knowledge of statistics. A median is the value at the middle of the list. The median time field would have the middle or fifth value in the last 11 blocks. Therefore, an odd number is a pre-requisite for items in a list. The list mtime must always have 11 members.

The time field value stored in the timelh variable is added to the mtime list for the first 11 blocks. When the loop variable, blkno or block number has a value of 11 or higher, the 0th element or the first time is removed and the time of the current block is added at the end. This ensures that the mtime list meets the condition of exactly 11 members, at all points in time. This approach removes the oldest time from the start of the list and adds the latest time to the end of the list, keeping the list size constant at 11.

For the first 2000 blocks, this method worked as a charm and then the median time in the block started going off track. Nevertheless, it remains as the median time of the time values of the last 11 blocks. The first check is on the middle or median time in the list of 11 blocks.

When errors surfaced, the list was created dynamically which ultimately ended with the list of 11 previous blocks. Yes, there is a better way of creating the list T!!! The list of times, T is calculated dynamically. The variable timem which is the median time of the block, lies in this list. Though, it may not be the median but it lies in this list of the last 11 times.

The value in the errors variable increments by 1 only when the current mediantime field is not the middle or median time in the last. The else statement is never called.

This median field is not stored in the block but the Bitcoin code calculates it. Most of the blocks median values are the middle value in the last 11 blocks.

Finally, out of a total of 453891, only 30679 blocks gave an error where the fifth block in the list of 11 prior blocks did not represent the median value. Only about 10% of the blocks were erroneous. Once again, the else is not called.

In October 2017, we get a block count of 489756 and an error count of 31162.

Create a table called miners before running the next program.

```
create table miners (blkno integer, miner varchar(2048))
```

Adding Names of the Miners into a Table and then Ranking them

```
ch2806.py
import struct
import psycpg2
import json
import subprocess
def cleanascii(text):
    return ''.join(i for i in text if ord(i)<128 and ord(i) >= 32)
conn = psycpg2.connect("dbname='postgres' user='postgres' host='localhost'
password='accord'")
cur = conn.cursor()
for blkno in range(1, 500000):
    print "block Number %d" % blkno
    s1 = "Select hash from blockhash where blkno = %d" % (blkno )
    cur.execute(s1)
    row = cur.fetchone()
    hash = row[0]
    raw = subprocess.check_output(["bitcoin-cli" , "getblock" , hash])
```

```

#print raw
raw = json.loads(raw)
version = raw["version"]
#print version
txid = raw["tx"][0]
#print "coinbase txid %s" % txid
rawt = subprocess.check_output(["bitcoin-cli", "getrawtransaction", txid, "1"])
#print rawt
rawt = json.loads(rawt)
coinbase = rawt["vin"][0]["coinbase"]
coinbase = coinbase.decode('hex')
#print coinbase
#print coinbase.encode('hex')
cleandata = ""
if ord(coinbase[0:1]) == 3 :
    str1 = coinbase[1:4] + "\x00"
    height = struct.unpack("I", str1)[0]
    if height == blkno:
        print "passed",
    else:
        print "failed height %d blkno %d" % (height, blkno),
    coinbase = coinbase[4:]
    cleandata = cleanascii(coinbase)
    if cleandata.find("Mined by f2pool") >= 0:
        cleandata = "Mined by f2pool"
    elif cleandata.find("Mined by AntPool") >= 0:
        cleandata = "Mined by AntPool"
    elif cleandata.find("BitFury") >= 0:
        cleandata = "BitFury"
    elif cleandata.find("BTCC") >= 0:
        cleandata = "BTCC"
    elif cleandata.find("BW Pool") >= 0:
        cleandata = "BW Pool"
    elif cleandata.find("KnCMiner") >= 0:
        cleandata = "KnCMiner"
    elif cleandata.find("slush") >= 0:
        cleandata = "slush"
    elif cleandata.find("slush") >= 0:
        cleandata = "slush"
    elif cleandata.find("21 Inc") >= 0:
        cleandata = "21 Inc"
    elif cleandata.find("Eligius") >= 0:
        cleandata = "Eligius"
    elif cleandata.find("BitClub Network") >= 0:
        cleandata = "BitClub Network"
    elif cleandata.find("Telco 214") >= 0:

```

```
cleandata = "Telco 214"
elif cleandata.find("BitMinter") >= 0:
cleandata = "BitMinter"
elif cleandata.find("P2Pool") >= 0:
cleandata = "P2Pool"
elif cleandata.find("Eobot") >= 0:
cleandata = "Eobot"
elif cleandata.find("1hash") >= 0:
cleandata = "1hash"
elif cleandata.find("Kano") >= 0:
cleandata = "Kano"
elif cleandata.find("haobtc") >= 0:
cleandata = "haobtc"
cleandata = cleandata.replace("“", "")
print cleandata
s1 = "insert into miners values (%d , '%s') " % (blkno , cleandata)
cur.execute(s1)
conn.commit()
```

Output

```
block Number 453890
passed BW Pool
block Number 453891
passed Mined by AntPool
```

In October 2017, the last block mined was block number 489756.

The first transaction in every block, the Coinbase transaction was always ignored. The input of the Coinbase transaction must be ignored; the input of the genesis block was newspaper headline!!!

The outputs of the Coinbase transaction have a count of Bitcoins made by the miner. It started with 50 Bitcoins and then 25 BTC and now a measly 12.5, it keeps reducing by half every 210,000 blocks. In October 2017, we have already mined 80% of all Bitcoins. You can read many interesting Bitcoin trivia at the website <http://www.bitcoinblockhalf.com>.

The blockchain explorer blockchain.info shows that block 454299, vintage February 23rd, 2017 was paid a block reward of 12.5 Bitcoins and from the transaction fees, the miner earned only 1.80012075 BTC. At some point in time, the miners will make money only from transaction fees paid to them.

The Coinbase input now has an entire BIP devoted to it, BIP-34. This BIP introduced a new version type 2 for transactions. If there is anything new, then document it in a BIP.

It also states that the first byte of the input of the Coinbase transaction will be the number 3 and then the next 3 bytes will be the height of the transaction. After that, it's a fair game and miners can place whatever they want in the Coinbase input. The only condition to satisfy is that the data must start with a specific number for the number of bytes to be pushed on the stack. Three bytes can represent the heights for the next 150 years. By then, Bitcoin will not break, but if it does, they will then simply change the version number to say version 165.

Back to code. The `getblock` command retrieves the block details, starting from block no. 1 and not 0. The version number of the block is stored in a variable called `version`. Only the transaction hash of the Coinbase transaction is accessed. The field `tx` is a list of transactions present in this block. But, the 0th transaction in this list is stored in a variable called `txid`.

The `getrawtransaction` command with the Coinbase transaction hash fetches the Coinbase transaction details, which is stored in variable `rawt`.

The Coinbase input is accessed using the 0th member of the vin list and then the Coinbase field. The Coinbase string value is in raw bytes, so it is decoded. Try it manually, and look at the first vin member, it has a field called Coinbase. Seeing is believing.

Now the first byte of the Coinbase input data is checked. If it has a value of 3, then the Coinbase transaction is BIP-34 compliant. For example, block 1 has one transaction hash only starting with 0e3e and the Coinbase input starts with a 04. Such Coinbase transactions are ignored.

The slice operator is used to read the height of the transaction; it is stored in this input. Only 3 bytes represent the integer, so a trailing 00 is added as the `unpack` function wants 4 bytes to read a string. We could have individually multiplied the bytes by orders of 2, but too much effort from our side. These bytes are then compared with the block number. Generally, they are same but at times they are not. This is not enough to flag an error and stop. A Coinbase transaction input error by an overexcited miner who wins a lottery is not to be taken seriously. Not all bip's are mandatory.

Once these 4 bytes are read, they are removed from the Coinbase string. Now we have a problem. We have no knowledge on the structure of the remaining bytes in the input. The BIP-34 is silent. Google is silent. Maybe there is no pattern.

Still, to get some clarity we create a function `cleanascii`. This function removes all the nonprintable characters in the Coinbase string. So, the only characters kept are the ones which are greater than 32, a space, and less than 128. This is also called the printable ASCII set. Anything above 128 is a kludge. Not too proud of code like this, but it does the job and helps understand the concept.

The data has names of the largest miners in the Bitcoin world. Some of the bigger names are f2pool, AntPool etc. This list keeps changing over time. Some of the miners add Mined by before their name, some like BitFury keep it plain and simple. We simply add the filtered Coinbase string. Everything after the first 4 bytes in a Coinbase input is non-standard.

The `cleandata` value and the corresponding block number is then added to the miners table. Once all data is added to the table, run the following SQL code.

```
select miner , count(*) from miners group by miner order by 2 DESC limit 20;
```

miner	count
	214842
Mined by AntPool	22782
slush	13883
BTCC	13709
BW Pool	10118
BitFury	9924
Eligius	9481
KnCMiner	6290
BitMinter	5473
/P2SH/	2237
BitClub Network	2134
Kano	1727
haobtc	1115

```
Mined By ASICMiner | 1097
Mined by f2pool    | 675
Mined by BTC Guild | 554
S                  | 89
P                  | 77
Q                  | 73
T                  | 52
(20 rows)
```

The single largest lot of miners are the ones with an empty string or the name of the miner in the initial Bitcoin blocks. These blocks are ignored because they do not have the first byte as 3.

```
postgres=# select count(*) from miners1;
count
-----
453891
(1 row)
```

The order of miners as per our 453891 blocks is: AntPool is the largest, followed by slush, and then BTCC. When you run the same code, the miners ranking will change. This list was created in the year 2016. In October 2017, AntPool was yet first with 29040 block mined, followed by BTCC with 16464 and then slush at 15825.

Once the block number is obtained, we can access the date, thus it makes it easy to rank miners by month and give them an award. There is no correct way of identifying the miner who mined the block. The BIP should have standardized this or the miners themselves should have done it. The advantage is that someone could track the mining community. All that we know is that clear majority of miners are in China or is that also a Bitcoin myth.

The getblocktemplate Command

```
ch2807.py
import subprocess
import json
raw = subprocess.check_output(["bitcoin-cli", "getblocktemplate"])
raw = json.loads(raw)
print raw["height"]
fee = 0
for i in range(0, len(raw["transactions"])):
    fee = fee + raw["transactions"][0]["fee"]
print "fee %d" % fee
print len(raw["transactions"])
```

```
Output
454301
fee 964500000
1929
454301
fee 1014000000
2028
```


One command only for miners, is the `getblocktemplate` command. Please display the entire output, as it gives a dynamic display of how a block is created. This output of this command could fill an entire book.

The output shows that the height of the transaction is 454301 or 489724. This block is going to be mined but it has not yet entered the system. An error is displayed when we immediately ran

```
bitcoin-cli getblockhash 454301
error code: -8
error message:
Block height out of range
```

Better still, key in this block number 454301 in the blockchain explorer and it says no such block. However, reduce the block number by 1 and you will see a block with no next block hash on the upper right of the screen. This block has not been mined still. To do this you must be quick on your feet.

As a gentle reminder, do not use our block numbers as these blocks have been mined, use the block number you get.

The number of transactions keep changing, so the calculated fee also changes. The fee value keeps changing depending on the number of transactions. We will revisit this method `getblocktemplate` in the second last chapter.

We want to disclose the different means or methods used by a miner to calculate miner's fee per block. One fee is the block reward, which is 12.5 Bitcoins today. Then the miner also gets a fee per transaction, which is the inputs minus the outputs. Given below is one more option.

Getting the Miner's Fee from blockchain.info Website

```
create table fees (blkno integer , fee double precision);
```

```
ch2808.py
import psycopg2
import bitcoin
import json
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost'
password='accord'")
cur = conn.cursor()
cur1 = conn.cursor()
for blkno in range(400296, 402666):
    s1 = "Select hash from blockhash where blkno = %d" % (blkno )
    cur.execute(s1)
    row = cur.fetchone()
    if row is None:
        break
    hash = row[0]
    url =
'https://blockchain.info/block/%s?format=json&api_key=pleasegetyourownkeyandp
astehere' % (hash)
    print url
    data = bitcoin.make_request(url)
    #print data
    data = json.loads(data)
```

```
fee = data["fee"]
print fee
fee = fee * 1.0 / 100000000
print fee
s2 = "insert into fees values (%d , %3.8f)" % (blkno , fee)
print s2
cur1.execute(s2)
conn.commit()
```

Output

```
https://blockchain.info/block/0000000000000000665a591af5039002f09fdb7c96d8
86b7bfce77b8806d129?format=json&api_key=pleasegetyourrownkeyandpastehere
31727891
0.31727891
insert into fees values (400138 , 0.31727891)
```

```
https://blockchain.info/block/0000000000000000101d01c79783aeaf19aea09f69b7
33811d8df1664a70401?format=json&api_key=pleasegetyourrownkeyandpastehere
22027198
0.22027198
insert into fees values (400139 , 0.22027198)

postgres=# select count(*) from fees;
count
-----
140
(1 row)
```

The Bitcoin core does not have a command which gives access to the total fees paid to a miner per transaction or per block. So, we resort to external APIs like the one given by blockchain.info. You could use pip to install blockchain or follow our commands.

In the program, we create a URL with the command to be executed. We add the word block followed by the block number and for a json output, we add a key called format with a value of json. Also add a key called api_key, whose value is a number sent via email by the blockchain people. We add the wrong key intentionally.

The website people are doing a great job in maintaining a copy of the blockchain and giving free access to it. We simply sent an email to them and within hours we got the api key.

The make-request function from pybitcointools sends this request. The fee field will have the fee paid for the block.

We have to download the entire block to extract just one field. This takes lots and lots of bandwidth, we are happy to check with just 140 blocks.

The fee is multiplied by 1.0 and then divided by a 1 with 8 0's, as we calculated the last time and then the data is inserted into the fees table.

Since a floating-point number is inserted into the table, make sure that the field has %f format as there will many decimal places in the value.

Checking the Output Returned by the getnetworkhashps command

```

ch2809.py
import subprocess
import json

raw = subprocess.check_output(["bitcoin-cli" , "getbestblockhash"])
bestblock = raw[:-1]
print "Best Block %s" % bestblock

raw = subprocess.check_output(["bitcoin-cli" , "getblock" , bestblock])
raw = json.loads(raw)
time = raw["time"]
prevblockhash = raw["previousblockhash"]
print "Previous block hash %s" % prevblockhash
chainwork1 = raw["chainwork"]

maxtime = time
mintime = time
lookup = 3
for i in range (0,lookup):
    raw = subprocess.check_output(["bitcoin-cli" , "getblock" , prevblockhash])
    raw = json.loads(raw)
    time1 = raw["time"]
    maxtime = max(maxtime , time1)
    mintime = min(mintime , time1)
    print "MinTime %d MaxTime %d" % (mintime , maxtime)
    prevblockhash = raw["previousblockhash"]
    print "Previous Hash %s" % prevblockhash
    chainwork = raw["chainwork"]

print "chainwork first %s" % chainwork1
print "chainwork second %s" % chainwork

workdiff = int(chainwork1 , 16) - int(chainwork, 16)
print "Work Diff %x" % (workdiff)

timediff = maxtime - mintime
print "Time Diff %d" % timediff
ans = (workdiff * 1.0) / timediff
print "ans %E" % ans

raw = subprocess.check_output(["bitcoin-cli" , "getnetworkhashps" , str(lookup)])
raw = raw[:-1]
print raw
ans1 = float(raw)
print "ans1 %E" % ans1
print str(ans) == str(ans1)

```

Output

Best Block


```
ans1 1.138450E+19  
True
```

Our goal is to explain nearly every command supported by the program bitcoin-cli. This program moves closer to achieving it.

Once again, there is really no substantial code that makes up the code base of the bitcoin-cli client. Most of the activity happens in bitcoind, the Bitcoin server.

The `getbestblock` command returns the hash value of the best block according to our bitcoin server. The site blockchain.info gives this block a height or block number of 454305. However, this is the last block according to blockchain.info. At the time of writing this book, this block has no Next Block hash. But it will change by the time you try it.

A better definition from the wiki is the most difficult to recreate chain or the blockchain. You can view this as the longest chain or the current active chain.

The blocks received are not stored to disk immediately, they are stored in the cache or memory. Therefore, it is not wise to shut down the server by killing the program, a graceful exit is the need of the day.

The `getblock` command is called with the best block hash value to obtain the previous blocks hash value as well as the time the best block or the current block was mined. This time is stored in the 80-byte header. You can verify the previous block hash value with the explorer.

The most important field is the current chainwork field as it divulges the number of hashes consumed to reach this height in the blockchain. This is not an actual number of hashes but a theoretical value. Chainwork has been covered earlier.

The start and end time is stored in variables, `maxtime` and `mintime`. They are initialized to the time of the best block. The objective is to calculate not the number of hashes consumed between two blocks, but the number of hashes consumed per second between two blocks.

The lookup variable is set to 3, thus it requires the network hash rate per second from the current best block. The hashes consumed can be computed from a certain block as well.

The loop iterates 3 times in our case, as the lookup variable has a value of 3. In the loop, the details of the previous block are retrieved. Here the hash is stored in the variable `prevblockhash`. At the end of the loop, the value in the `prevblockhash` variable is changed and it points to the previous block field stored in the header of this block. We are basically moving backwards in time across blocks. If the field we used was `nextblock`, we would be moving in the forward direction in the blockchain.

The time from the blockheader is assigned to the `time1` variable. Then an indirect if condition checks if the time in the `time1` variable is larger than the `maxtime` value, the current block time. If yes, then the `maxtime` variable is set to the current block time using the `max` function. Similarly, if the current block time stored in `time1` is smaller than the current block time in our variable `mintime`, the `mintime` variable is set to the current block time using the `min` function. We are not using the if statement.

The chainwork is also saved at the end of the loop, but its value is not used during the loop. The net effect is that the `chainwork1` variable is the best block or the hash of at the beginning. Variable `chainwork` is a block that is at an offset of 3 blocks away.

The next task is to compute the number of hashes that were consumed to create these 3 back to back blocks. This is achieved by simply subtracting the value in variable `chainwork1` from `chainwork`. The `chainwork1` value is larger than `chainwork` since we are moving backwards in the blockchain. The `int` function is used since both variables are strings.

The variables `chainwork` value would be larger if we were moving forwards. The `timediff` variable gives the difference in

time between the best block and the block, 3 blocks away. This method is used, strictly adhering to the norms given in the Bitcoin Core source code. Since we are moving backwards in time, the block creation time reduces.

The timediff variable divulges the time elapsed between the 3 blocks, the maxtime is subtracted from mintime. The workdiff variable is the difference between the two chain works. We then divide the workdiff value by the timediff value. This is how we know the network hash rate, the number of hashes calculated divided by the time elapsed. The multiplication by 1.0 is a Python standard.

The answer is displayed in a scientific notation, where the E signifies the number of 0s. A very large number.

The getnetworkhashps command is used with the lookup or number of blocks. The starting height can also be given; but the default is best block.

This command returns the network hashes per second in a scientific notation. The numbers are compared in string format because floating points numbers will never be equal.

The network hash rate has grown over time from 1.179825E+18 in mid 2016 to 4.145615E+18 in February 2017 to 1.138450403172016e+19 in October 2017. Most people run the getnetworkhash command, say every 10 minutes and store the output in a database. Then they pontificate on the continuous growth of bitcoin mining network.

Once again, it looks simple when you read the source. There is no other way to write this code.

The getrawmempool Command

```
ch2810.py
import subprocess
import json
raw = subprocess.check_output(["bitcoin-cli", "getrawmempool"])
raw = json.loads(raw)
print "Number of transactions %d" % len(raw)
for i in range(0,5):
    print "Transaction Hash %s" % raw[i]
    raw1 = subprocess.check_output(["bitcoin-cli", "getrawtransaction", raw[i], "1"])
    #print raw1
```

Output

```
Number of transactions 12300
Transaction Hash
e3c4b643f391e5ad7abae5a4035c3b0146f0f2626d12dc8c986d9bd607810000
Transaction Hash
7e3de77a4acf2b1524663539cf327678c67846f25aea5d8bb33d98044b100300
Transaction Hash
a6382b5447902544acc68bde60b74817ae5d0ce3211548b10cecf48c1aa20500
Transaction Hash
0c020dcfe73a41bdae07cce7c78b92f8ab845482545e05bfba86c4fe25fc0600
Transaction Hash
725b558ec3926c4d40b24187eaf685ce5582ca9fc105ff3c77345b709cb60700
Number of transactions 11344
Transaction Hash
7e3de77a4acf2b1524663539cf327678c67846f25aea5d8bb33d98044b100300
Transaction Hash
```

```

0c020dcfe73a41bdae07cce7c78b92f8ab845482545e05bfba86c4fe25fc0600
Transaction Hash
ff29e9007855c94a18c2d374ece98e3b5f9a22123583f55f87f178fdb2720a00
Transaction Hash
e7f6a5172ed4086b2a6e0477bf72c491420a9489334696225dbd5204c1890e00
Transaction Hash
a573be36df97ce636b9bbc7887090e60860cecdcefa4745df0d91aacb301a00

```

Any node or peer or miner that gains knowledge of a new transaction will broadcast the same to all the node/nodes it is connected to. Miners place these transactions in blocks, but these blocks are still to be mined so they remain unconfirmed. There is chaos everywhere.

What should a node do with transactions that are not confirmed? All these transactions are placed in a memory pool called mempool. The `getrawmempool` command returns a list of transaction hashes, that are present in the local mempool. Each computer has a unique mempool.

When we executed this program twice over an interval of 15 minutes, the number of transactions in the mempool changed each time. In October 2017, we actually got a whopping 21000 mempool transactions. We will keep coming back to the mempool.

The `getrawmempool` command returns a list of transaction hashes in the mempool. We take the transaction id starting with 725b and paste it into the `blockchain.info` browser. You must be very quick or else it will become a confirmed transaction. At times, it may display an un-confirmed transaction.

The `blockchain.info` browser first shows this transaction in red color signifying it is unconfirmed, and then informs us that the transaction has not been placed in any confirmed block to date.

After 10 minutes or so, this transaction has 2 confirmations and a block number of 426943. If you try this transaction hash today, it shows the block number being the same, but the confirmations will be in tens of thousands. For some mempool transactions, `blockchain.info` rejects them completely as all our mempool's will be very different.

After some time, our bitcoin server receives a valid block filled with lots of transactions. The bitcoin server removes all the transactions from the mempool, which are already present in the block it recently downloaded. New transactions get added and confirmed transactions get removed. Therefore, the mempool is in such a great flux. A transaction that goes into the mempool is thoughtfully frisked for errors from head to toe.

```

create table trandata (blkno bigint, size bigint , notran bigint, noinputs bigint,
nooutputs bigint, coinoutputs bigint, coin varchar(1024) , coin1 varchar(1024) ,
fees double precision, miner double precision);

```

The curious case of a transaction hash starting with

```
91546e98f5350052a0f910efbb37a19444254cac7748474b20fce5e3735439c9.
```

For some time, no block was willing to accept this transaction. Finally, it got accepted. It is still not known when a transaction hash will be removed from the mempool.

Saving Every Detail of a Transaction into a Database

```
create table trandatao (blockno integer , baddr varchar(128));
```

```

ch2811.py
from cfuncs2 import *
import leveldb

```

```
import psycopg2
import pybitcointools
import struct
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost'
password='accord'")
cur = conn.cursor()
cur1 = conn.cursor()
cur2 = conn.cursor()
errors = 0
totalcoins = 0

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def doutput1(f , index):
    bitcoinvalue = r8bytes(f)
    outputscripflen = rvarint(f)
    scripkey = rbytes(f , outputscripflen).encode('hex')
    return bitcoinvalue

def dinput1(f ):
    prevtrhash = rhash(f).encode('hex')
    outputindex = rint(f)
    inputscripflen = rvarint(f)
    sigkey = rbytes(f , inputscripflen).encode('hex')
    seqno = rint(f)

def getoutput(thash , outputindex):
    #print thash
    hash1 = '74' + reversehash(thash)
    v = db.Get(hash1.decode('hex'))
    (fileno , offset) = base128(v , 0)
    (offsetb , offset) = base128(v , offset)
    (offsett , offset) = base128(v , offset)
    f1 = open("/Users/vijaymukhi/Library/Application
Support/Bitcoin/blocks/blk%05d.dat" % fileno, "rb")
    f1.seek(offsetb + 80 + offsett , 0)
    tver = rint(f1)
    noinputs = rvarint(f1)
    totoutputs = 0
    for i in range ( 0 , noinputs):
        dinput1(f1 )
    nooutputs = rvarint(f1)
    for i in range ( 0 , nooutputs):
        (outputb) = doutput1(f1 , outputindex)
    if i == outputindex:
        f1.close()
        return outputb
    locktime = rint(f1)
```



```

def doutput(f , tno , blkno):
    miner = 0
    coinoutputs = 0
    nooutputs = rvarint(f)
    if tno == 0:
        coinoutputs = nooutputs
        totboutputs = 0
    for i in range ( 0 , nooutputs):
        bitcoinvalue = r8bytes(f)
        if tno == 0:
            miner = miner + bitcoinvalue
        #print "miner %d bitcoinvalue %d noutputs %d" % (miner , bitcoinvalue ,
        nooutputs)
        if tno >= 1:
            totboutputs = totboutputs + bitcoinvalue
        #print "--(%d) totboutputs %d bitcoinvalue %d" % (tno , totboutputs , bitcoinvalue)
        outputscriptlen = rvarint(f)
        scriptpubkey = rbytes(f , outputscriptlen)
        if tno ==0 and bitcoinvalue != 0 and outputscriptlen >= 16:
            #print "ScriptPubKey %s:%d" % (scriptpubkey.encode('hex') ,
            outputscriptlen)
            baddr = ""
            if outputscriptlen == 67:
                #print "Bitcoin address %s" %
                pybitcointools.pubkey_to_address(scriptpubkey[1:-1])
                baddr = pybitcointools.pubkey_to_address(scriptpubkey[1:-1])
            elif outputscriptlen == 35:
                #print "Bitcoin address %s" %
                pybitcointools.pubkey_to_address(scriptpubkey[1:-1])
                baddr = pybitcointools.pubkey_to_address(scriptpubkey[1:-1])
            elif outputscriptlen == 25:
                #print "Bitcoin address %s" %
                pybitcointools.script_to_address(scriptpubkey)
                baddr = pybitcointools.script_to_address(scriptpubkey)
            elif outputscriptlen == 23:
                #print "Bitcoin address %s" %
                pybitcointools.script_to_address(scriptpubkey)
                baddr = pybitcointools.script_to_address(scriptpubkey)
            else:
                print "Bitcoin address %s" %
                pybitcointools.script_to_address(scriptpubkey)
            print "Script %s" % scriptpubkey.encode('hex')
            print "outputscriptlen %d" % outputscriptlen
            exit()
        s1 = "insert into trandata values(%d , '%s')" % (blkno , baddr)
        cur2.execute(s1)
    return (nooutputs , coinoutputs , totboutputs , miner)

```

```
def dinput(f , tno , ver ,blkno ):
    global errors , totalcoins
    coin = ''
    noinputs = rvarint(f)
    totalbitcoins = 0
    for i in range ( 0 , noinputs):
        prevtrhash = rhash(f)
        outputindex = rint(f)
        if tno > 0:
            bitcoins = getoutput(prevtrhash[::-1].encode('hex') ,outputindex )
            totalbitcoins = totalbitcoins + bitcoins
            #print "(%d)totalbitcoins %d:%f bitcoins %d number of inputs %d" % (tno ,
            totalbitcoins , (1.0 * totalbitcoins) / 100000000 , bitcoins , noinputs)
            inputscripflen = rvarint(f)
            sigpubkey = rbytes(f , inputscripflen)
            str3 = ''
            if tno == 0 and ord(sigpubkey[0:1]) == 3 and ver == 2 :
                totalcoins = totalcoins + 1
                str1 = sigpubkey[1:4] + "\x00"
                oddoreven = 0
                height = struct.unpack("I" , str1)[0]
                if blkno % 2 == 1:
                    oddoreven = 0
                height = height + oddoreven
                if height != blkno:
                    errors = errors + 1
                    print "Error in height %d" % blkno
                if tno == 0:
                    coin = ''
                    i = 0
                    length = len(sigpubkey)
                    while length > 0:
                        if ord(sigpubkey[i:i+1]) >= 32 and ord(sigpubkey[i:i+1]) <= 127:
                            coin = coin + sigpubkey[i:i+1]
                            length = length - 1
                            i = i + 1
                    seqno = rint(f)
                    return (noinputs, coin , '' , totalbitcoins)

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
```

```

else:
    return (n,offset)

def index(ans):
    (nVersion , offset) = base128(ans , 0)
    (nHeight, offset) = base128(ans , offset)
    (nStatus, offset) = base128(ans, offset)
    (nTx, offset) = base128(ans, offset)
    (nFile, offset) = base128(ans, offset)
    (nDataPos, offset) = base128(ans, offset)
    return (nFile , nDataPos)

blockno = 0
transactions = 0
sizetran = 0
tranmax = 0
tranmin = 12345678
db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index')
for blkno in range(1, 5000000):
    if blkno % 1000 == 0:
        print blkno
        s1 = "Select hash from blockhash where blkno = %d" % (blkno )
        cur.execute(s1)
        row = cur.fetchone()
        try:
            hash = row[0]
        except:
            print "hash is null"
            break
        hash = reversehash(hash)
        hashb = "62" + hash
        hashb = hashb.decode('hex')
        try:
            ans = db.Get(hashb)
        except:
            print "Over and out"
            break
        (nFile , nDataPos ) = index(ans)
        f = open("/Users/vijaymukhi/Library/Application
        Support/Bitcoin/blocks/blk%05d.dat" % nFile, "rb")
        f.seek(nDataPos - 4 , 0)
        size = rint(f)
        header = f.read(80)
        (ver , prevhash , merklehash , time1 , diff , nonce) = struct.unpack("I32s32sIII" ,
        header[0:80])
        notran = rvarint(f)
        transactions = transactions + notran

```

```
totnoinputs = 0
totnooutputs = 0
totnobininputs = 0
totnoboutputs = 0
coinoutputs = 0
miner = 0
for tno in range ( 0 , notran):
    startfp = f.tell()
    tver = rint(f)
    (noinputs,coin , coin1 , onebininputs) = dinput(f , tno , ver , blkno )
    if tno == 0:
        #print "coin %s" % coin
        coin = coin.replace("“", "")
        totnobininputs = totnobininputs + onebininputs
        #print "totnobininputs %d onebininputs %d" % (totnobininputs , onebininputs )
        (nooutputs, dummycoinoutputs , oneboutputs , dummyminer ) = doutput(f ,
        tno , blkno)
        if tno == 0:
            miner = dummyminer
            coinoutputs = dummycoinoutputs
        if tno != 0:
            totnoinputs = totnoinputs + noinputs
            totnooutputs = totnooutputs + nooutputs
            totnoboutputs = totnoboutputs + oneboutputs
        locktime = rint(f)
        endfp = f.tell()
        transize = endfp - startfp
        if transize > tranmax:
            tranmax = transize
        if transize < tranmin:
            tranmin = transize
        sizetran = sizetran + transize

    #print "File number %d Block Number %d size of data %d size transaction %d no
    tran %d total size %d average %d %d %d" % (nFile , blkno , size , transize ,
    transactions , sizetran , sizetran / transactions , tranmin , tranmax )
    #print "Total Outputs %d Total Inputs %d fees %d" % (totnoboutputs ,
    totnobininputs , totnoboutputs - totnobininputs )
    #print "totnoboutputs %d totnobininputs %d" % (totnoboutputs , totnobininputs)
    fees = totnobininputs - totnoboutputs
    #print "fees %d" % fees
    #print "miner coinbase %d" % miner
    s2 = "insert into trandata values(%d , %d , %d , %d , %d , %d , '%s' , '%s' , %f , %d) "
    % (blkno , size , notran , totnoinputs , totnooutputs , coinoutputs , coin , coin1 ,
    (fees * 1.0) / 100000000 , miner)
    print s2
    cur1.execute(s2)
```

```

conn.commit()
f.close()
print "Total Number of transactions %d Total Size size %d average %d %d %d"
% (transactions , sizetran , sizetran / (1.0 * transactions) , tranmin , tranmax)
print "Total Number of transactions %d Total Size size %d average %d" %
(transactions , sizetran , sizetran / transactions)
print "Total Coins %d errors %d" % (totalcoins , errors)

```

We ran this code first in mid-2016.

Output

```

Error in height 199686
Error in height 209920
Error in height 216460
Total Number of transactions 148223612 Total Size size 79494214725 average 536
62 999657

Total Number of transactions 148,223,612
Total Size size 79494214725 average 536
Total Coins 140547 errors 3

```

After running the program for over a week, we get only 3 errors. We ran though 148 million transactions.

```

postgres=# select count(*) from trandata;
count
-----
400001
(1 row)

select count(*) from trandatao;
count
-----
2244030
(1 row)

```

This program takes a very long time to run. We explain why.

The for loop in the program starts from block number 1 and goes all the way to block 5000000 that does not exist. The hash value of every block is stored in the blockhash table. A value of 0x62, 'b' is added to the hash, so we finally know where it starts on disk. The 80-byte block header is read.

Now it is time to analyze every transaction using our own code.

The file pointer is at the start of the number of transactions contained in each block. This count is saved in a variable called notran. The variable transactions store a running total of all the transactions in all the blocks. This number is displayed at the end of the program, more to disclose the number of transactions that have occurred since the genesis block was created.

The dinput function handles all the inputs for a single transaction and returns 4 values. In this function, the varying number of inputs is read in a variable, noinputs and it is returned.

There is an if statement to check if we have moved past the Coinbase transaction. This event will only happen when the

tno variable is larger than 1. The first transaction in a block is always the Coinbase transaction where the inputs are ignored. The function called `getoutput` takes the transaction hash found in the input and the output index and returns the Bitcoin amount present in that output.

This function `getoutput` simply adds a 0x74 to the reverse of the transaction hash and gets the file number and position, where this transaction starts using the index folder. Then it reads the amount field of the desired output and returns this value back. No asking for data from a web server.

The bitcoins brought in by this input is added to the `totalbitcoins` variable and it is returned as the last parameter. If it is a Coinbase transaction, the first byte is checked for 3 and the height of the version number is checked to be 2. Just an ignorable error check.

Then once again, if it is a Coinbase transaction, the name of the miner is extracted using a crude while loop. This miner's name is stored in the `coin` variable and it is returned. A lot of the code here works only with a Coinbase transaction. The `coin1` variable is ignored if the value is an empty or null string. Any single quotes are replaced with double quotes in the miner's name. The variable `totnoinputs` gives a running total of all the amounts the inputs bring into all the transactions.

The `doutput` function handles all the outputs of a single transaction. First, the number of outputs is read in a variable called `nooutputs`, this value is returned later.

Next, the variable called `coinoutputs` is assigned a value which is the number of outputs present in a Coinbase transaction. The variable `totboutputs` stores the total Bitcoin value of all outputs minus the outputs that go to a miner. The miner's outputs are returned in a separate variable called `miner`.

A simple if statement that checks for the value of variable `tno` being 0 or 1 and more, solves the problem. Then the script public key value is read in greater detail. Depending upon the size of the length of this field, the bytes are extracted from this field. The function `script_to_address` from the `pybitcointools` is smart enough to return the Bitcoin address.

```
select * from trandatao limit 2;
blockno | baddr
-----+-----
1       | 12c6DSiU4Rq3P4ZxiKxziL5LmMBrzjrJX
2       | 1HLoD9E4SDFFPDYfNYnkBLQ85Y51J3Zb1
```

This output reveals that block number 2 has a Bitcoin address starting with 1HL, which is confirmed by `blockchain.info` also. By adding the block number and Bitcoin address to the `trandatao` table, a list of blocks and Bitcoin addresses is obtained. We could have had a separate table to store every output and its Bitcoin address, or simply removed the if condition `tno == 0`.

Once we finish dealing with inputs and outputs, it's time to update our running totals. It is handled very differently for Coinbase transactions. The `miner` variable holds the number of Bitcoins the miner gets. The `coinoutputs` is the number of outputs. For a non-Coinbase transaction the running totals of all the inputs and outputs is saved. The running totals of the Bitcoin values found in the inputs and outputs is updated too.

The size of data occupied just by the transactions is determined and the minimum and maximum transaction size are stored in two variables, `tranmax` and `tranmin`. Some of these variables are stored as per block number and they will be displayed when all gets over.

This code is basically all previous programs put together as one large program. Now you get the gist of what we are going to do, be a statistician's delight and throw up all sorts of numbers at you. We are trying not to use `bitcoin-cli` program as it is very slow.

In the year 2017 and henceforward, this program will take a lot more time. You will be scanning more than 200 million transactions. We did not run this program in October, 2017.

Checking if our Calculated Fees Match with the Output of the Blockchain Website

```
ch2812.py
import psycopg2
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost'
password='accord'")
cur = conn.cursor()
cur1 = conn.cursor()
for blkno in range (400000, 402667):
    s1 = "Select * from fees where blkno = %d" % (blkno )
    cur.execute(s1)
    row = cur.fetchone()
    s2 = "Select * from trandata where blkno = %d" % (blkno )
    cur1.execute(s2)
    row1 = cur1.fetchone()
    if row is None or row1 is None:
        print "Row is None %d" % blkno
        exit()
    if row[1] - row1[8] < 0.00001 :
        print "Block Number %d Passes" % blkno
        print row
        print row1
        print row[1] , row1[8] , row[1] - row1[8]
    else:
        print "block Number %d Fails" % blkno
        exit()
```

Output

```
Block Number 400138 Passes
(400138, 0.31727891)
(400138L, 749156L, 1577L, 3814L, 3467L, 1L, '', '', 0.317279, 2531727891.0)
0.31727891 0.317279 -8.99999999526e-08
Block Number 400139 Passes
(400139, 0.22027198)
(400139L, 999980L, 1398L, 5513L, 2897L, 1L, '', '', 0.220272, 2522027198.0)
0.22027198 0.220272 -1.99999999895e-08
Row is None 400140
```

Finally, a small program to validate our actions with the data entered in two tables in the last programs.

The fees table has only two fields, the first column is block number and the second column is called fee. The value in fee is obtained from the blockchain website. The trandata table has many fields, the ninth column/field is called fees. The fee has been calculated using the earlier example.

We cross-check the values of the second field of the fees table with the ninth field of the trandata table. If the fees

match, we are on track. Now you can slice and dice the data as you wish. Checking just 140 out of half a million is not enough, but hey we are not Coinbase!

It is very odd that in blocks like 48002 and 48009 and more, we see that the miner has taken no fees at all. We wondered how was this possible. Then we realize that this block only contained a coinbase transaction.

Proof that this is not an isolated incident.

```
select * from fees where fee <= 0;
```

blkno		fee
480002		0
480009		0
480014		0
480026		0
480066		0

CHAPTER 29

fee_estimates.dat

Let's get the bad news out of the way. The Bitcoin framework needs a way of determining how much fee to pay a miner for carrying our transactions. It used lots and lots of statistics to come up with an algorithm to calculate the fees. For some reason, they made a very complex fee calculation even more complex. What follows will not work with Bitcoin Core Version 0.15 but it will work with the other hard forks.

But, please read this chapter. We have an older fee_estimates.dat along with the source code files. You can use the newer fee_estimates.dat created by Bitcoin Core Version 0.15 also. See for yourselves where the changes have been made.

The Bitcoin Core doubles up like a wallet. It is a full-fledged wallet. In the Bitcoin world, we pay a small fee to the miner to accept our transaction. Miners are also very choosy. They pick transactions that have a high fee. It simply proves that there is no standardization on the fee.

When a wallet is used to send a transaction, the fee is not specified by us, the sender. The wallet calculates the miner fee internally. So, the question here is how do the wallets compute this fee. The answer lies in a file called fee_estimates.dat. Again, Google has no information on the data stored in this file, our only source was the Bitcoin source code.

This chapter examines this smallest file, fee_estimates.dat in the Bitcoin ecosystem. It is also one of the most difficult file for anyone to comprehend.

We copy this file, fee_estimates.dat on our Desktop. It is not an important file for Bitcoin. So, if the file is deleted or not present, the Bitcoin server at startup will simply re-create this file. This file is written to when the Bitcoin server shuts down. Observe the data time stamp on this file and you will realize that it gets updated when the server is shutdown.

Let's now look at the data in the fee_estimates.dat file and understand a method, estimatefee which uses this data. The objective of this chapter is to simulate the workings of the estimatefee method and achieve the same result. The estimatefee method gives an estimate of the fee to be paid to a miner to place our transaction at a certain position in the block. We will never understand the statistics used in estimating a fee.

```
bitcoin-cli estimatefee 2
0.00051920
```

When we run the same command a little later

```
bitcoin-cli estimatefee 2
0.00082971
```

The command estimatefee takes a parameter with a value in the range of 2 and 25. The number given here is the relative position in the blockchain where we want to place our block. In our case, we want to know the fee we should pay to the miners to include our transaction within 2 blocks. With time, the amount paid to the miners also changes. There is a slight increase, 0.0003. Today, October 2017 the same fee increases by an order of magnitude, it is 0.00241668.

```
bitcoin-cli estimatefee 3
0.00043906
```

The output shows a lower fee as the transaction must be included within 3 blocks. The key word here is within. We are willing to wait for one block to be mined to pay less, but it's all in the hands of the miner gods, we are simply asking for a estimate.

```
bitcoin-cli estimatefee 26
-1

bitcoin-cli estimatefee 1
-1
```

Bitcoin does not keep transaction statistics beyond 25 blocks. The minimum number of blocks must be 2 as the key word here is a block within a range, 2 - 25. An answer of -1 signifies that it is out of range.

Reading the Version Number from the File fee_estimates.dat

```
ch2901.py
from cfuncs import *
f = open("/Users/vijaymukhi/Desktop/fee_estimates.dat")
ver = rint(f)
print "Version %d" % (ver)
verc = rint(f)
print "Version Client %d" % (verc )
```

Output
Version 109900
Version Client 120100

The same code after upgrading our client.

Output
Version 109900
Version Client 130100

Finally, for version 0.15 we get the Version as 149900 and Version Client as 159900.

This file has 2 version numbers. The first version number is hard-coded as seen in the source code. This offending line in the source code looks like: fileout << 109900

There is a variable fileout in the C++ source code, which is overloaded with the << operator to write something. The value is written to a file, in this case. The value is hardcoded to 109900 which is rarely seen.

The second version is our client version. When we move to the next version 0.13.01, the version number changes but the format of the file, fee_estimates.dat file is the same.

Reading the Next Two Fields, Height and Decay

```
ch2902.py
from cfuncs import *
def fdouble(f):
    return struct.unpack('d', f.read(8))[0]
```

```
f = open("/Users/vijaymukhi/Desktop/fee_estimates.dat")
ver = rint(f)
print "Version %d" % (ver)
verc = rint(f)
print "Version Client %d" % (verc )
height = rint(f)
print "height %d" % height
decay = fdouble(f)
print "%f" % decay
```

Output

```
Version 109900
Version Client 120100
height 427372
0.998000
```

The third field in the file fee_estimates.dat is the height of the blockchain. The only way to confirm this output is by checking the last block number with the blockchain.info website.

The next field or the fourth field is called decay and it is a number with decimal places. This field has a constant value of 0.998. In the program, the fdouble function reads 8 bytes from the file and then unpacks them as a decimal or floating point number. This number is again a hard-coded value in the Bitcoin source. The fee_estimates.dat file has many hard-coded numbers of double/float type.

We run the same code later and the height changes.

Output

```
height 485659
0.0000
```

Here, we stopped checking the newer version as you will notice that the decay is one of the hearts of the fee estimation code.

The Relation Between Half-Life and 346 Blocks

```
ch2903.py
fee = 1000.0
for i in range ( 0 , 348):
    print " (%d) %f" % (i , fee) ,
    fee = fee * 0.998
```

Output

```
(0) 1000.000000 (1) 998.000000 (2) 996.004000
(345) 501.229639 (346) 500.227180 (347) 499.226725
```

Let's now understand what this number, .998 is all about. Only the first three values and the last three values of the output are displayed.

Let's take a fee of 1000. The half of 1000.0 is 500.0.

When 1000.0 is multiplied by 0.998, the result is a smaller value. The number 1000 now becomes 998.00. This number

9998.0 is again multiplied by 0.998 and it gives 996.004. After 346 multiplications, its value is 500.227180; the original value reduces to half. This value is called a half-life.

The objective is to find out how many iterations it takes to reach half the value.

Let's assume that 346 represents the number of blocks to be created. We know that on an average, a block is normally created after 10 minutes. So, 6 blocks take an hour and 144 blocks are created in a day. Taking this logic forward 346 blocks will take about 2.4 days to create.

If we keep multiplying a number by the decay or a value of 0.998, the value will reduce to half over 346 iterations or over 2.4 days. By reducing its value, its weight in the calculations also reduces.

Introducing the Concept of a Bucket to Hold Fees

```
ch2904.py
bucketBoundary = 1000.0
FEE_SPACING = 1.1
COIN = 100000000
MAX_MONEY = 21000000 * COIN
MAX_FEERATE = 1e7
cnt = 1
while ( True):
    print "(%d) %f" % (cnt , bucketBoundary)
    cnt = cnt + 1
    bucketBoundary = bucketBoundary * FEE_SPACING
    if bucketBoundary >= MAX_FEERATE:
        break

print "(%d)%f" % ( cnt , MAX_MONEY)
print 1e7
```

Output

```
(1) 1000.000000
(2) 1100.000000
(3) 1210.000000

(96) 8556676.046608
(97) 9412343.651269
(98)2100000000000000.000000
10000000.0
```

The variables in the program are named in this way more to mirror the source code. The source code is in C++ and we are converting it to Python code. Any variable in all caps is a #define or a macro in C++, it is one way of representing a constant value in C/C++.

Let us assume that the minimum fee in BTC paid per Kb for transaction data is 1000.0. The variable bucketBoundary is the main variable that represents a Bitcoin fee. Some more variables are created and then there is an infinite while loop. The counter variable, cnt increases by 1 each time. The values in cnt and the bucketBoundary variable are displayed in the loop.

We would like to represent various type of fees spread over many fees starting with 1000 BTC per KB. The minimum fee is multiplied with 1.1 or the value in the variable FEE_SPACING.

In the loop, the fee is 1000 in first round. The second time, the fee changes to 1100 BTC-per-KB. The fee gets larger and larger with every iteration.

This can go on forever, so there is an if statement to check if the fee or the variable bucketBoundary is larger than what Bitcoin believes is a fee that no one will ever pay, 10,000,000 or 1e7. This is the value of the MAX_FEERATE variable. The source code does not offer any explanation of why this range value.

In our code, we check at what count, the fee will reach the largest fee that we can pay. The answer is when the cnt variable reaches a value of 97. The counting starts from 1 and not 0.

The value of the variable COIN is the Satoshi multiplication number, 1 with 8 zeroes.

In Bitcoin, a maximum of 21,000,000 Bitcoins can be created. This value is converted into Satoshi to represent the largest Bitcoin number ever seen. Therefore, it is called MAX_MONEY in Satoshi and not Bitcoin.

Displaying the fee buckets stored in the file fee_estimates

```
ch2905.py
from cfuncs import *
def fdouble(f):
    return struct.unpack('d', f.read(8))[0]
f = open("/Users/vijaymukhi/Desktop/fee_estimates.dat")
ver = rint(f)
print "Version %d" % (ver)
verc = rint(f)
print "Version Client %d" % (verc )
height = rint(f)
print "height %d" % height
decay = fdouble(f)
print "%f" % decay
noofbuckets = ord(f.read(1))
print "Number of buckets %d" % noofbuckets
for i in range (1 , noofbuckets + 1):
    print "(%d) %f" % (i , fdouble(f))
print
```

Output

```
Version 109900
Version Client 130100
height 447932
0.998000
Number of buckets 98
(1) 1000.000000
(2) 1100.000000
(3) 1210.000000

(96) 8556676.046608
(97) 9412343.651269
(98) 2100000000000000.000000
```

This program explains the workings of the earlier program. The next field stored in the fee_estimates.dat file is the

number of buckets. A bucket is nothing but a container. Here it stores fees. Buckets are assigned ranges of fees. For example, Bucket 1 stores fees in the range of 1000 to 1099. Bucket 2 stores in the range of 1100 to 1209, etc. So, a fee of say 1150 BTC per KB will be stored in the bucket 2 because 1150 is larger than 1100 but less than 1210. The value displayed in the output is the miner fee value as per the buckets.

Any transaction that has a fee larger than 1100 and less than 1210 will be stored in bucket 2, assuming the count starts from 1 and not 0.

The number 98 is once again fixed as seen in the previous example on half-life. These 98 fee ranges or buckets are stored in the file even though they will never ever change. These numbers, bucket number and the values stored in them are constants.

The program reads the values stored in each bucket. The first value 98 is a count of the buckets following. In C++ code, this data is stored using a list or an array or a vector. In C++, arrays/vectors start with a count of the members following.

The last value is to take care of all values set larger than the maximum fee set by Mr. Satoshi.

The source code has some files that exclusively mention Mr. Satoshi, some of them at times mention the Bitcoin Core developers. This means that a lot of code originally written by Mr. Nakamoto has been untouched by human hands. True genius.

The fee_estimates.dat File

```
ch2906.py
from cfuncs import *
def double(f):
    return struct.unpack('d', f.read(8))[0]
bucketlist = []
fileavglist = []
txctvavglist = []
conflisttemp = []
finalconflist = []
f = open("/Users/vijaymukhi/Desktop/fee_estimates.dat")
ver = rint(f)
#print "Version %d" % (ver )
verc = rint(f)
#print "Version Client %d" % (verc )
height = rint(f)
#print "height %d" % height
decay = double(f)
#print "decay %f" % decay
noofbuckets = ord(f.read(1))
print "Number of buckets %d" % noofbuckets
for i in range (0 , noofbuckets):
    temp1 = double(f)
    bucketlist.append(temp1)
    #print " %f" % temp1,
    nofileavg = ord(f.read(1))
    print "FileAvg %d" % nofileavg
    for i in range (0 , nofileavg):
```

```

    temp2 = double(f)
    fileavglist.append(temp2)
    #print "(%d) %f " % ( i , temp2) ,
noTxCtAvg = ord(f.read(1))
print "TxCtAvg/TotalNum %d" % noTxCtAvg
for i in range (0 , noTxCtAvg):
    temp3 = double(f)
    txctvavglist.append(temp3)
    #print "(%d) %f " % ( i , temp3) ,
    #print "Where %d" % f.tell()
rows = ord(f.read(1))
print "rows %d" % rows
for row in range(0 , rows):
    #print "nConf Row is %d" % row
    conflisttemp = []
    columns = ord(f.read(1))
    print "columns %d" % columns
    for col in range(0 , columns):
        temp5 = double(f)
        conflisttemp.append(temp5)
        #print "(%d) %1.6f" % (col , temp5) ,
        #print
    finalconflist.append(conflisttemp)

```

Output

```

Number of buckets 98
FileAvg 98
TxCtAvg/TotalNum 98
rows 25
columns 98

```

This program basically displays almost nothing as all the print statements are commented out. The fee_estimates.dat file now contains two sets of arrays or lists. The first is for fees and the second one is for the priorities of transactions. The first change in the program is that the fee ranges/buckets are read in a list called bucketlist. These are hard coded values representing fees paid for transactions.

The original code is written in C++, so to represent the same in python, the vector or an array has the length first and then the actual data. The double value stored in the array is read in a temporary variable named temp1 using the double function. Then these double types are added to the appropriate list. Every list has the same size of 98 members.

The next list is unarguably the most important list as it stores the actual fee paid per transaction. Each transaction in a block pays a certain fee. This fee is added to the appropriate bucket. The second list is called fileavglist and the third list is called txctvavglist.

There is also a moving average of fees. The source code first multiplies the stored values by .998 and then adds the current value. In this manner, we downgrade or decay the previously stored average. This is the only bucket where the actual fee is stored.

The next bucket type stores a simple count of every transaction that contributes a fee to the bucket. If the bucket

contains a value of 50, then it is assumed that 50 transactions had fees in the range of that bucket. Once again, we decay the count of the transactions by multiplying by the decay. The values are stored in the last two buckets. In our code, we do not decay the values, but the source code does it. The second list has larger values than the third one.

The crux of the matter is that old transactions do not contribute significantly to the values stored in the bucket. The half-life is 2.4 days.

Now comes a more complex list, a two-dimensional list. The number of rows is 25. This is a hard-coded value in the source code and therefore blocks beyond 25 cannot be tracked. Our fee estimation stops at 25 blocks. The rows are stored sequentially. For the columns, first the number of columns is stored and then the column data. In this case, the column data has a constant size, 98.

This concept is a little unique. The idea is that if a transaction with a certain fee is confirmed within say 4 blocks, then it is also confirmed within 5 blocks and within 15 blocks and so on and so forth.

In other words, let's say a transaction is confirmed within 2 blocks with a certain fee. The bucket number 2 must be increased as this bucket represents the fee the transaction paid. Simultaneously there must be an increase in all rows larger than 2, i.e. 2, 3, 4 up to 25. This is because the same transaction that got confirmed within two blocks, also got confirmed within 3 blocks. A row and a block is synonymous here. It cannot be just one row, all rows larger than the current one will also be incremented.

Thus, the `conflisttemp` list is a list like the `txCtAvg` list as they both contain the count of transactions. The difference is that it contains a count of all the transactions that have been mined by paying a fee within that many blocks.

To paraphrase from a comment in the source, we are counting the total number of transactions that were confirmed within Y blocks in each bucket or fee range.

The Estimatefee Command Output Matches the Fee Computed from `fee_estimates.dat` File

```
ch2907.py
from cfuncs import *
import subprocess
def fdouble(f):
    return struct.unpack('d', f.read(8))[0]
bucketlist = []
fileavglist = []
txctvavglist = []
conflisttemp = []
finalconflist = []
f = open("/Users/vijaymukhi/Desktop/fee_estimates.dat")
ver = rint(f)
verc = rint(f)
height = rint(f)
decay = fdouble(f)
noofbuckets = ord(f.read(1))
for i in range (0 , noofbuckets):
    temp1 = fdouble(f)
    bucketlist.append(temp1)
nofileavg = ord(f.read(1))
#print "FileAvg"
for i in range (0 , nofileavg):
```

```

    temp2 = fdouble(f)
    #print "(%d) %f " % (i , temp2) ,
    fileavglist.append(temp2)
#print "noTxCtAvg"
noTxCtAvg = ord(f.read(1))
for i in range (0 , noTxCtAvg):
    temp3 = fdouble(f)
    #print "(%d) %f " % (i , temp3) ,
    txctvavglist.append(temp3)
rows = ord(f.read(1))
#print "Conf"
for row in range(0 , rows):
    conflisttemp = []
    columns = ord(f.read(1))
    for col in range(0 , columns):
        temp5 = fdouble(f)
        #print "(%d) %f " % (col, temp5) ,
        conflisttemp.append(temp5)
        #print
    finalconflist.append(conflisttemp)
#print "sufficientTxVal / (1 - decay) %f" % (1/ (1 - decay))
#for confTarget in range(2 , 26):
for confTarget in range(2 , 3):
    nConf = 0.0
    totalNum = 0.0
    curNearBucket = bestNearBucket = bestFarBucket = noofbuckets - 1
    for bucket in range ( noofbuckets - 1 , 1 , -1):
        nConf = nConf + finalconflist[confTarget - 1][bucket]
        totalNum = txctvavglist[bucket] + totalNum
        print "(%d) totalNum %f condition %s" % (bucket , totalNum , totalNum >= 1.0 / (1 - decay))
        if totalNum >= 1.0 / (1 - decay) :
            curPct = nConf / (totalNum )
            print "(%d) nConf %f curPct %f break condition %s" % (bucket , nConf , nConf /
            (totalNum) , curPct < 0.95)
            if curPct < 0.95:
                break
            else:
                nConf = 0
                totalNum = 0
                bestFarBucket = bucket
                bestNearBucket = curNearBucket
                curNearBucket = bucket - 1
                print "(%d) bestNearBucket %d bestFarBucket %d" % (bucket ,
                bestNearBucket , bestFarBucket)
                median = -1.0
                txSum = 0.0

```

```
minBucket = min(bestNearBucket , bestFarBucket )
maxBucket = max(bestNearBucket , bestFarBucket )
print "minbucket %d maxbucket %d" % (minBucket , maxBucket)
for j in range (minBucket , maxBucket + 1):
    txSum += txctvavglist[j]

txSum = txSum / 2
for j in range (minBucket , maxBucket + 1) :
    print "(%d) txctvavglist[j] %f txSum %f condition %s" % (j , txctvavglist[j] ,
    txSum , txctvavglist[j] < txSum)
    if txctvavglist[j] < txSum:
        txSum = txSum - txctvavglist[j]
    else:
        median = fileavglist[j] / txctvavglist[j]
        break

raw = subprocess.check_output(["bitcoin-cli" , "estimatefee" , str(confTarget)])
raw = float(raw[:-1]) * 100000000
median = int(median)
print "confTarget %02d raw %5.0f median %5.0f ans %f:%s" % (confTarget , raw , median ,
median - raw , (median - raw <= 0.0000000001))
```

Output

```
43) totalNum 41364.644319 condition True
(43) nConf 39811.531924 curPct 0.962453 break condition False
(43) bestNearBucket 43 bestFarBucket 43
(42) totalNum 9504.857667 condition True
(42) nConf 8969.585348 curPct 0.943684 break condition True
minbucket 43 maxbucket 43
(43) txctvavglist[j] 41364.644319 txSum 20682.322159 condition False
confTarget 02 raw 56826 median 56816 ans -10.000000:True
```

This program and the next couple of programs are a precursor to understanding or deciding the fees to pay a miner, based on their past fees.

Finding the median fee paid to a miner is quite simple and easy as all that is required is dividing the average fee with the number of transactions that have contributed to this fee. The only problem lies in choosing the bucket.

Coming back to our code, the bucketlist list decides on the bucket depending upon the transaction fee. This list has constant values.

The confTarget variable represents the number of blocks. Change the number 3 to number 26 and rest assured, the blocks until 25 match the fee returned by the command estimatefee. These changes are made in the first for statement only. Uncomment the comment and see for yourself. Most of the code has been explained in the earlier programs.

Let's take a simple case where the variable confTarget has a value of 2 only.

The two variables, nConf and totalNum are initialized to 0 for every new block. There are 3 bucket variables which are set to 97, one less than the total number of fee buckets. The variable noofbuckets will always have a fixed value of 98.

The bucket with the largest fee is taken up first. Therefore, the inner for loop starts from 97 or one less than the total number of buckets. The loop goes backwards, from a higher fee to a lower fee. The idea is to attain the lowest fee to be

paid. The fees will be lower with lower buckets. At a breakpoint of 95%, once again a hard-coded value, the loop quits. Had we been looking for the highest fee to be paid, then the search would be from the start at the first bucket ignoring all the lower fees paid.

The for statement starts from an initial value of 97 and reduces by 1 each time.

The variable `nConf` stores the number of transactions. It also stores running values as a floating-point number. The `finalconflist` variable is a list of lists. This time, the number of blocks is given attention. The reason being, the bucket gives a fee range but the blocks ascertain the number of the blocks within the bucket, which are required for a confirmation.

There are two loops. One for loop has a variable `confTarget`, which has a range of blocks where a confirmation is obtained and the next for loop has a variable `bucket`, which starts from the highest to the lowest value. These two variables are used to index the double list, `finalconflist`.

The list `txctvavglist` stores the number of transactions in each fee range. A running total is maintained in the `totalNum` variable of the last bucket, counting downwards of the total number of transactions. The values of these variables computed in every iteration of the for loop is put out for display.

Not all data points in our bucket are included. A number 499 is hardcoded as the minimum number of data values a bucket must have, for a data point to be considered for inclusion. Where did we get this magic value of 499 from?

The condition $1.0 / (1 - \text{decay})$ gives a value of 499. The if statement will be true when the variable `totalNum` reaches 499 or more.

Once again, the `totalNum` variable is a running total; but it starts from the highest bucket value. This value acts as a filter to make sure that some of the buckets do not get involved into figuring out the median.

The inner loop ends when the percentage storing variable `curPct` drops to less than 95%. This percentage is calculated by dividing the `nConf` variable by the `totalNum` variable.

Initially the success percentage is very high. It then starts dropping. The minute it reaches a success rate of less than .95%, we are skating on thin ice and hence it is time to quit out. It makes no sense continuing further when the success percentage rate drops beyond what is acceptable. This range is hard coded in the code. It is a minimum of 95% for success. At the other end, failure is a measly 5%. For the last time, when a success rate of 95% is achieved, it's time to quit out.

If the percentage is above 95%, the variables `nConf` and `totalNum` are zeroed out and the bucket variables is reassigned to the far and near values. The biggest problem is that we have not been able to gather a data set where the near and far buckets are different.

Once the minimum and maximum bucket values are obtained, the variables `bestNearBucket` and `bestFarBucket` are used. These variables are only initialized in the else part of the percentage if statement.

The total count of transactions between the two bucket values are added. A for loop is used which loops only once as the variables `minBucket` and `maxBucket` have the same value. The index variable named `j` is obtained by using the `minBucket` variable as the starting point and `maxBucket + 1` as the ending point. The `txctvavglist` gives the list value at position 43 in this case only. The value in the `txSum` variable is divided by half after the first for loop.

When the for loop is entered again, there is an if statement which checks if the list `txctvavglist` value is less than the value in the `txSum` variable, if yes, the value of `txSum` is changed. If no, then we have hit the jackpot and it's wise to quit out.

But first the median is calculated. The loop is not significant here. The median is calculated by diving the actual average with the count of transactions. To verify our computation, the `estimatefee` command is used and the values are then compared. As these are floating point numbers, it is a tedious task to compare them for equality.

The objective is not calculating the median but the average fee of the best bucket that met our condition of 95% success.

To meet success with this program, you must be quick on your feet. This file, `fee_estimates.dat` gets updated only when we shut down the bitcoin server. So, shut down, copy the file and start the server again. Then run this program. You cannot use our `fee_estimates.dat` in this program.

Understanding the Priority Buckets

```
ch2908.py
bucketboundary = 100000000 * 144 / 250
PRI_SPACING = 2
MAX_PRIORITY = 1e16
cnt = 0
while ( True):
    print "(%d) %f" % (cnt , bucketboundary) ,
    bucketboundary = bucketboundary * PRI_SPACING
    cnt = cnt + 1
    if bucketboundary >= MAX_PRIORITY:
        break
COIN = 100000000
MAX_MONEY = 21000000 * COIN
INF_PRIORITY = 1e9 * MAX_MONEY
print "%f" % INF_PRIORITY
```

Output

```
(0) 57600000.000000
(1) 115200000.000000
1932735283200000.000000 (26) 3865470566400000.000000 (27)
7730941132800000.000000 21000000000000000.000000
```

The earlier program looked at the fee bucket, now there are similar buckets for priorities. The priorities buckets are however smaller in number, only 29 of them. The priority bucket starts with 57600000. This number is calculated by dividing 144 by 250 and then multiplying by 10^{**8} . The Spacing is not 1.1 as it was for fees, but 2. The maximum priority is a very large number 1 with 16 zeroes.

The for loop quits when a priority is reached. The last priority number obtained is `MAX_MONEY` multiplied by 1 with 9 zeroes.

Reading the Priority Objects Stored in the File `fee_estimates.dat`

```
ch2909.py
from cfuncs import *
import subprocess
def fdouble(f):
    return struct.unpack('d', f.read(8))[0]
bucketlist = []
fileavglst = []
txctvavglst = []
```

```

conflisttemp = []
finalconflist = []
f = open("/Users/vijaymukhi/Desktop/fee_estimates.dat")
ver = rint(f)
verc = rint(f)
height = rint(f)
decay = fdouble(f)
noofbuckets = ord(f.read(1))
for i in range(0, noofbuckets):
    temp1 = fdouble(f)
    bucketlist.append(temp1)
nofileavg = ord(f.read(1))
#print "FileAvg"
for i in range(0, nofileavg):
    temp2 = fdouble(f)
    #print "(%d) %f " % (i, temp2),
    fileavglist.append(temp2)
#print "noTxCtAvg"
noTxCtAvg = ord(f.read(1))
for i in range(0, noTxCtAvg):
    temp3 = fdouble(f)
    #print "(%d) %f " % (i, temp3),
    txctvavglist.append(temp3)
rows = ord(f.read(1))
#print "Conf"
for row in range(0, rows):
    conflisttemp = []
    columns = ord(f.read(1))
    for col in range(0, columns):
        temp5 = fdouble(f)
        #print "(%d) %f " % (col, temp5),
        conflisttemp.append(temp5)
    #print
    finalconflist.append(conflisttemp)

decay = fdouble(f)
print decay
noofbuckets = ord(f.read(1))
print "Priority Buckets"
for i in range(0, noofbuckets):
    temp1 = fdouble(f)
    print "(%d) %f " % (i, temp1),
    bucketlist.append(temp1)
nofileavg = ord(f.read(1))
#print "FileAvg"
for i in range(0, nofileavg):
    temp2 = fdouble(f)

```

```
#print “(%d) %f “ % (i , temp2) ,
fileavglist.append(temp2)
#print “noTxCtAvg”
noTxCtAvg = ord(f.read(1))
for i in range (0 , noTxCtAvg):
    temp3 = fdouble(f)
    #print “(%d) %f “ % (i , temp3) ,
    txctvavglist.append(temp3)
rows = ord(f.read(1))
#print “Conf”
for row in range(0 , rows):
    conflisttemp = []
    columns = ord(f.read(1))
    for col in range(0 , columns):
        temp5 = fdouble(f)
        #print “(%d) %f “ % (col, temp5) ,
        conflisttemp.append(temp5)
    #print
    finalconflist.append(conflisttemp)
print “Where %d” % f.tell()
```

Output

(0) 57600000.000000

0.998

(27) 7730941132800000.000000 (28) 2100000000000000125829120.000000 Where 28534

-rw---@ 1 vijaymukhi staff 28534 Sep 3

The program displays the second priority object.

The decay is 0.998, like the fees. Then there are the same buckets for the first object to store the priority ranges, a total of 29 of them. This is followed by the average priority and a count of transactions. Finally, at the end, there is the double list.

After reading the priority data, the position of the final pointer is displayed. This position helps determine the file size. The file pointer in the fee_estimates.dat file is at the end of the file. Both methods have a file size of 28534, which is the file size. No hash lurking at the end. No more data structures stored after it.

Checking the Priority with the Estimatepriority Command

```
ch2910.py
from cfuncs import *
import subprocess
def fdouble(f):
    return struct.unpack('d', f.read(8))[0]
bucketlist = []
fileavglist = []
txctvavglist = []
conflisttemp = []
```

```

finalconflist = []
f = open("/Users/vijaymukhi/Desktop/fee_estimates.dat")
ver = rint(f)
verc = rint(f)
height = rint(f)
decay = fdouble(f)
noofbuckets = ord(f.read(1))
for i in range(0, noofbuckets):
    temp1 = fdouble(f)
    bucketlist.append(temp1)
nofileavg = ord(f.read(1))
#print "FileAvg"
for i in range(0, nofileavg):
    temp2 = fdouble(f)
    #print "(%d) %f " % (i, temp2),
    fileavglist.append(temp2)
#print "noTxCtAvg"
noTxCtAvg = ord(f.read(1))
for i in range(0, noTxCtAvg):
    temp3 = fdouble(f)
    #print "(%d) %f " % (i, temp3),
    txctvavglist.append(temp3)
rows = ord(f.read(1))
#print "Conf"
for row in range(0, rows):
    conflisttemp = []
    columns = ord(f.read(1))
    for col in range(0, columns):
        temp5 = fdouble(f)
        #print "(%d) %f " % (col, temp5),
        conflisttemp.append(temp5)
        #print
    finalconflist.append(conflisttemp)

bucketlistp = []
fileavglistp = []
txctvavglistp = []
conflisttempp = []
finalconflistp = []

decay = fdouble(f)
noofbucketsp = ord(f.read(1))
for i in range(0, noofbucketsp):
    temp1p = fdouble(f)
    bucketlistp.append(temp1p)
nofileavgp = ord(f.read(1))
#print "FileAvg"
for i in range(0, nofileavgp):

```

```
temp2p = fdouble(f)
#print "(%d) %f " % (i , temp2p) ,
fileavglistp.append(temp2p)
#print "noTxCtAvg"
noTxCtAvgp = ord(f.read(1))
for i in range (0 , noTxCtAvgp):
    temp3p = fdouble(f)
    #print "(%d) %f " % (i , temp3p) ,
    txctvavglistp.append(temp3p)
rowsp = ord(f.read(1))
#print "Conf"
for rowp in range(0 , rowsp):
    conflisttemp = []
    columnsp = ord(f.read(1))
    for colp in range(0 , columnsp):
        temp5p = fdouble(f)
        #print "(%d) %f " % (col, temp5p) ,
        conflisttemp.append(temp5p)
        #print
    finalconflistp.append(conflisttemp)
for confTarget in range(2 , 26):
    nConf = 0.0
    totalNum = 0.0
    curNearBucket = bestNearBucket = bestFarBucket = noofbucketsp - 1
    for bucket in range ( noofbucketsp - 1 , 1 , -1):
        nConf = nConf + finalconflistp[confTarget - 1][bucket]
        totalNum = txctvavglistp[bucket] + totalNum
        #print "(%d) totalNum %f condition %s" % (bucket , totalNum , totalNum >=
        1.0 / (1 - decay))
        if totalNum >= 0.2 / (1 - decay) :
            curPct = nConf / (totalNum )
            #print "(%d) nConf %f curPct %f break condition %s" % (bucket , nConf , nConf /
            (totalNum) , curPct < 0.95)
            if curPct < 0.95:
                break
            else:
                nConf = 0
                totalNum = 0
                bestFarBucket = bucket
                bestNearBucket = curNearBucket
                curNearBucket = bucket - 1
                #print "(%d) bestNearBucket %d bestFarBucket %d" % (bucket ,
                bestNearBucket , bestFarBucket)
                median = -1.0
                txSum = 0.0
                minBucket = min(bestNearBucket , bestFarBucket )
```



```

maxBucket = max(bestNearBucket , bestFarBucket )
#print "minbucket %d maxbucket %d" % (minBucket , maxBucket)
for j in range (minBucket , maxBucket + 1):
    txSum += txctvavglistp[j]

txSum = txSum / 2
for j in range (minBucket , maxBucket + 1) :
    #print "(%d) txctvavglist[j] %f txSum %f condition %s" % (j , txctvavglist[j] ,
    txSum , txctvavglist[j] < txSum)
    if txctvavglistp[j] < txSum:
        txSum = txSum - txctvavglistp[j]
    else:
        if txctvavglistp[j] == 0:
            median = -1
        else:
            median = fileavglistp[j] / txctvavglistp[j]
        break

raw = subprocess.check_output(["bitcoin-cli" , "estimatepriority" , str(confTarget)])
raw = float(raw[:-1]) * 100000000
if raw == -100000000:
    raw = -1
median = int(median)
print "confTarget %02d raw %5.0f median %5.0f ans %f:%s" % (confTarget , raw ,
    median , median - raw , (median - raw <= 0.0000000001))

```

Output

```

confTarget 02 raw -1 median -1 ans 0.000000:True
confTarget 03 raw -1 median -1 ans 0.000000:True
confTarget 04 raw -1 median -1 ans 0.000000:True
confTarget 05 raw -1 median -1 ans 0.000000:True

```

This program calculates the priority required for a transaction to be part of a block. We had high hopes but our answer was always -1. The reason being, a zero-fee transaction has no hope in hell also of entering the blockchain. So, if you do not want to pay up, no miner will include your transaction. On paper, at least a zero-transaction fee has a chance. Though this can be misused, as seen in the case of Ethereum.

Now change the condition, replace the 1.0 with a smaller value 0.2. The priority list in consideration also changes, but not the fee lists. The code otherwise is the same.

Please run the same set of commands to confirm the above.

```

bitcoin-cli estimatepriority 2
-1

bitcoin-cli estimatepriority 25
-1

```

Let's now look at another; estimatesmartfee.

```

bitcoin-cli estimatefee 1
-1

```

```
bitcoin-cli estimatefee 2
0.00056826
```

The fee to be paid for confirming a transaction within 1 block is -1 as there is not enough data points for Bitcoin Core to take a call. However, we know the fee to be paid if we wanted a confirmation within 2 blocks.

```
bitcoin-cli estimatefee 2
0.00056826

bitcoin-cli estimatesmartfee 1
{
  "feerate": 0.00056826,
  "blocks": 2
}
```

The command estimatesmartfee continuously calls the function that calculates the median. It increases the number of blocks by 1 each time until it gets a median larger than 0 or till it reaches the maximum of 25 blocks.

The feerate member enlists the fee to be paid and the block member discloses within how many blocks the transaction will get confirmed.

Bitcoin core 0.15 has removed the estimatepriority method from its arsenal.

Checking with the Estimatesmartfee Command

```
ch2911.py
from cfuncs import *
import json
import subprocess
def fdouble(f):
    return struct.unpack('d', f.read(8))[0]
bucketlist = []
fileavglist = []
txctvavglist = []
conflisttemp = []
finalconflist = []
f = open("/Users/vijaymukhi/Desktop/fee_estimates.dat")
ver = rint(f)
verc = rint(f)
height = rint(f)
decay = fdouble(f)
noofbuckets = ord(f.read(1))
for i in range (0 , noofbuckets):
    temp1 = fdouble(f)
    bucketlist.append(temp1)
nofileavg = ord(f.read(1))
#print "FileAvg"
for i in range (0 , nofileavg):
    temp2 = fdouble(f)
    #print "(%d) %f " % (i , temp2) ,
```

```

        fileavglist.append(temp2)
    #print "noTxCtAvg"
    noTxCtAvg = ord(f.read(1))
    for i in range(0, noTxCtAvg):
        temp3 = fdouble(f)
        #print "(%d) %f " % (i, temp3),
        txctvavglist.append(temp3)
    rows = ord(f.read(1))
    #print "Conf"
    for row in range(0, rows):
        conflattemp = []
        columns = ord(f.read(1))
        for col in range(0, columns):
            temp5 = fdouble(f)
            #print "(%d) %f " % (col, temp5),
            conflattemp.append(temp5)
            #print
        finalconflat.append(conflattemp)

    #print "sufficientTxVal / (1 - decay) %f" % (1 / (1 - decay))
    #for confTarget in range(2, 26):
    def cmedian(confTarget):
        nConf = 0.0
        totalNum = 0.0
        curNearBucket = bestNearBucket = bestFarBucket = noofbuckets - 1
        for bucket in range(noofbuckets - 1, 1, -1):
            nConf = nConf + finalconflat[confTarget - 1][bucket]
            totalNum = txctvavglist[bucket] + totalNum
            if totalNum >= 1.0 / (1 - decay):
                curPct = nConf / (totalNum)
                if curPct < 0.95:
                    break
                else:
                    nConf = 0
                    totalNum = 0
                    bestFarBucket = bucket
                    bestNearBucket = curNearBucket
                    curNearBucket = bucket - 1
                    txSum = 0.0
                    minBucket = min(bestNearBucket, bestFarBucket)
                    maxBucket = max(bestNearBucket, bestFarBucket)
                    for j in range(minBucket, maxBucket + 1):
                        txSum += txctvavglist[j]

                    txSum = txSum / 2
                    for j in range(minBucket, maxBucket + 1):
                        if txctvavglist[j] < txSum:

```

```
txSum = txSum - txctvavglist[j]
else:
    if txctvavglist[j] == 0:
        return -1
    else:
        return fileavglist[j] / txctvavglist[j]
for blockm in range( 1 , 26):
    block = blockm
    median = -1
    while median < 0 and block <= 25:
        median = cmedian(block)
        block = block + 1

    raw = subprocess.check_output(["bitcoin-cli" , "estimatesmartfee" , str(blockm)])
    raw = json.loads(raw)
    mediano = raw["feerate"] * 100000000
    if int(mediano) == int(median) and raw["blocks"] == block - 1 :
        print "(%d) Matches %d:%d %d:%d" % (blockm , int(mediano) ,int(median) ,
        raw["blocks"] , block - 1 )
    else:
        print "(%d) Does not Match %d:%d %d:%d" % (blockm , int(mediano) ,int(median) ,
        raw["blocks"] , block - 1 )
```

Output

```
(1) Matches 56826:56826 2:2
(2) Matches 56826:56826 2:2
(3) Matches 43943:43943 3:3
```

The code to compute the median in the earlier program is now placed in a function called `cmmedian`. The function returns the computed median or -1 when a median cannot be calculated. The code otherwise remains the same.

```
bitcoin-cli estimatesmartfee 0
{
    "feerate": -1,
    "blocks": 0
}
```

The program has a for statement. The block number starts with 1 as the smart fee gives a median of -1 at block number 0. In the while loop, the condition demands that the median is a positive number or the blocks are less than 25, the number of blocks. The median is calculated and the block number is increased by 1 with every iteration. It is like calculating the next median with the `estimatefee` command having the block number incremented by 1.

The objective is to return a median that is positive. The rest of the code simply checks if our results match up with the return value of the `estimatesmartfee` command, both in terms of fee rate and blocks; the block variable is subtracted by -1.

We can replicate the source code methodology and get a minimum fee from the mempool.

The smart priority is ignored as the answer is always -1.

Let's calculate the priority of a transaction whose hash is :

9486322d4476097b747e28e504dd7ab4da42ecb2d0c62874362730e06ead53c0.

Calculating the Priority of an Actual Transaction

```
ch2912.py
import subprocess
import json
txid = '9486322d4476097b747e28e504dd7ab4da42ecb2d0c62874362730e06ead53c0'

def tdetails(txid , oindex):
    raw = subprocess.check_output(["bitcoin-cli" ,"getrawtransaction" , txid , "1" ])
    #print raw
    raw = json.loads(raw)
    btc = raw["vout"][oindex]["value"]
    btc = int(btc * 100000000)
    blockhash = raw["blockhash"]
    raw1 = subprocess.check_output(["bitcoin-cli" ,"getblock" , blockhash ])
    raw1 = json.loads(raw1)
    bheight = raw1["height"]
    return (btc,bheight)

raw = subprocess.check_output(["bitcoin-cli" ,"getrawtransaction" , txid , "1" ])
#print raw
raw = json.loads(raw)
tsize = raw["size"]
print "transaction Size %d" % tsize
blockhash = raw["blockhash"]
print "BlockHash is %s" % blockhash
raw1 = subprocess.check_output(["bitcoin-cli" ,"getblock" , blockhash ])
raw1 = json.loads(raw1)
bheight = raw1["height"]
bheight = bheight - 1
print "Block Height is %d" % bheight
dResult = 0
inChainInputValue = 0
nTxSize = tsize
for i in range(0,len(raw["vin"])):
    txid = raw["vin"][i]["txid"]
    vout = raw["vin"][i]["vout"]
    print "Transaction Hash %s:%d" % (txid , vout)
    (btc,bheightt) = tdetails(txid , vout)
    diff = bheight - bheightt
    inChainInputValue = inChainInputValue + btc
    print "Output Bitcoins %d Block height %d:%d diff %d" % (btc, bheightt , bheight , diff)
    dResult = dResult + btc * diff
    print "dResult %d inChainInputValue %d" % (dResult , inChainInputValue)
```

```
scriptSig = raw["vin"][i]['scriptSig']['hex']
print "scriptSig %s" % scriptSig
length = (len(scriptSig) ) / 2
offset = 41 + min(110 , length)
print "Length of scriptSig %d offset %d nTxSize %d" % (length, offset , nTxSize)
if nTxSize > offset :
    nTxSize = nTxSize - offset
print "nTxSize %d" % nTxSize

print

entryPriority = 1.0 * dResult / nTxSize
deltaPriority = (-1.0 * inChainInputValue) / nTxSize
print "entryPriority %f deltapriority %f" % (entryPriority , deltaPriority)
totalpriority = entryPriority + deltaPriority
print "TotalPriority %f" % totalpriority
```

Output

```
transaction Size 372
BlockHash is
000000000000000000cc25277488c896d25a20a5b1df9315a6acdfb597988f82
Block Height is 427779
Transaction Hash
cb807f90fe600fe3020cdd22c840c2ebfcd2be62794f7e6d16a611080e5e2597:0
Output Bitcoins 40818919 Block height 427750:427779 diff 29
dResult 1183748651 inChainInputValue 40818919
scriptSig
47304402207af3f83906b2f382d319ad40448a02d07597521bd10ca8b951d218c5e0baed
49022017268b3d47ac65f4e793f52e3d54b41c2d3294add73a14b264e357aed7fae66e01
21026b7b9f366a002facaf91fb007da69944299927e9c7a038b552cf1c7b418d2547
Length of scriptSig 106 offset 147 nTxSize 372
nTxSize 225

Transaction Hash
3b310eb4eb1b7b94fb83432d2a20b74dd28fe36d21cc0523be912d87cf996175:0
Output Bitcoins 10713038 Block height 427723:427779 diff 56
dResult 1783678779 inChainInputValue 51531957
scriptSig
47304402207af3f83906b2f382d319ad40448a02d07597521bd10ca8b951d218c5e0baed
49022017268b3d47ac65f4e793f52e3d54b41c2d3294add73a14b264e357aed7fae66e01
21026b7b9f366a002facaf91fb007da69944299927e9c7a038b552cf1c7b418d2547
Length of scriptSig 106 offset 147 nTxSize 225
nTxSize 78

entryPriority 22867676.653846 deltapriority -660666.115385
TotalPriority 22207010.538462
```

Please look at the details of this transaction and the others using website blockchain.info and follow the above hard coded example. You can also uncomment the display of the raw variable and see the entire transaction. We have

displayed a few basic details about this transaction, which are acquired by calling the `getrawtransaction` command with the above transaction id and the last parameter as 1.

The field size shows that the transactions size is 372 bytes. The `getrawtransaction` command does not return the block number with this transaction but only the block hash value. This block hash value is then used to obtain the height of the block in the blockchain, 427780.

The priority of the transaction is calculated before it enters the blockchain. This transaction hash has only two inputs and two outputs.

The website `blockchain.info` does not display the transaction hash and the output index in the inputs. If you recall, the inputs points to the outputs that will bring in the Bitcoins to be transferred. This transaction hash had not yet been added to the blockchain when this program was executed. Today it is buried many blocks deep in the blockchain over 70,000 confirmations, time does fly.

The inputs are the Bitcoins owned by the user, which can be readily spent. These outputs are accessed in a very indirect manner. At the end of the day, the inputs and outputs deal only in Bitcoins.

The same code is used to obtain the following transaction hashes.

The first transaction that supplies inputs is

```
cb807f90fe600fe3020cdd22c840c2ebfcd2be62794f7e6d16a611080e5e2597
```

and the output index is 0. There is a function called `tdetails` that takes the transaction id and the output index and returns the amount of Bitcoins owned by this transaction and its height.

The same `getrawtransaction` command with this transaction hash gives the details of the transactions. The number of Bitcoins it brings to the table from the 0th output is 40818919 Satoshi. The height of this block is availed by using the `getblock` command. All this has been explained earlier.

This Bitcoin value is obtained by first accessing the `vout` list and thereafter, the output denoted by the `oindex` variable and then the `value` field. This number is multiplied by 100000000 to convert it into Satoshi. The website `blockchain.info` also confirms this value in BTC's. This transaction is however, in block number 427750. The `getblock` command on the `blockhash` field gives the number. The difference in blocks being 427779 - 427750, i.e. 29 is blocks stored in the `diff` variable. This is like ageing whiskey, the longer you age, the better the taste and the more you pay.

The concept of priority is very simple. All that it implies is, the older the inputs, the greater the priority to be given to the transaction. What it does not say offhand, is that the priority also increases with the value of the inputs. This makes sense as transactions which are have higher value for the inputs should gain precedence over transactions that are of lower value in Bitcoins. The rich get priority, an unfortunate fact of life.

The output in Bitcoins is multiplied by the difference in blocks i.e. $40818919 * 29$, which results in a value of 1183748651. This value is stored in a variable called `dResult` (just following the Bitcoin source code). There will be a sum of all the inputs in course of time.

Each output value in BTC is captured in another variable called `inChainInputValue`, which is 40818919 after the first for iteration. Both `dResult` and `inChainInputValue` are running totals, one only of Bitcoins and the other also of Bitcoins but aged over time.

At the End of the First Transaction Input.

```
dResult 1183748651
```

```
inChainInputValue 40818919
```

Now for the second transaction input. It's transaction hash is

3b310eb4eb1b7b94fb83432d2a20b74dd28fe36d21cc0523be912d87cf996175 and the output index is also fortunately 0.

It brings to the table 10713038 Satoshi. The block number that contains this transaction is 427723. The difference is $427779 - 427723$, 56 blocks. When the number of Satoshi 10713038 is multiplied with the block difference 56, the result is 599930128. This value is added to the value stored in the dResult variable. The answer is 1783678779. When the value 10713038 is added to the value already present in the inChainInputValue variable, the total is 51531957 Satoshi.

At the end of the second fetch, which is the last round as there are only two inputs in the original transaction, the result is

```
dResult 1783678779
inChainInputValue 51531957
```

The transaction hash in blockchain.info

9486322d4476097b747e28e504dd7ab4da42ecb2d0c62874362730e06ead53c0 shows the Total Input and it matches with the program calculated value, 51531957. This field is called Total Input.

Now for a small twist in the tale. It is important to calculate the modified size of the transaction. All fees and priorities have something to do with this modified size of the transaction and not with the actual size of the transaction. This transaction has a real or actual size of 372 bytes as returned by the getrawtransaction command. But this size is not used to compute the priority.

Now to something not done before. The length of the input signatures of the two inputs in our original transaction hash is calculated. The length of the scriptSig variable cannot be accessed directly. It is 106 bytes by accessing the 0'th input and then the scripSig field and finally the hex field. For some reason, in blockchain.info, the first byte 47 is not displayed. There is also an extra space separating the two values. In blockchain.info the input script is placed on two different lines, hence this extra byte.

A variable called offset is calculated in our code which has a constant value of 41 plus the size of the script signature but with a difference. The catch is that if the script signature is larger than 110 bytes, only the first 110 bytes are considered. In this case and in most cases, it will be lower than 110 bytes. This length is the value of the decoded scripts.

Where does the magic number 41 come from? This number has not been randomly thought off. It is a calculated value. The inputs use a 32-byte transaction hash. Another 4 bytes for the output index are added taking the total to 36 bytes. The unused sequence number takes up 4 bytes. This gives a total of 40 bytes. The length of the Script Signature data is assumed to take up 1 byte, it's length is a variable int. And voila, it is the magic number of $40 + 1 = 41$. This magic number 41 is the constant size of the input. This is how we understand every bit and not byte of a Bitcoin transaction.

The offset variable has a value of $41 + 106$, which is 147 bytes in this case and in most inputs except the multisig ones. The value of the original size of the transaction is 372 bytes and it is stored in the variable, tsize. This value in variable nTxSize is larger than the value of the offset variable. The value of the offset variable, 147 is reduced from the original value of nTxSize. So, the new value of the variable nTxSize is $372 - 147 = 225$.

In the second iteration of the inputs, the same process is followed and 147 is reduced from the value of 225. The modified size is 78 bytes. Both inputs have the same length of the signature, this is generally not a coincidence.

The value in dResult 1783678779 is divided by the modified size of the transaction, 78 bytes. The result is the value of priority, 22867676.653846. This value is called entryPriority. Another priority called deltaPriority is calculated, which is the variable inChainInputValue divided by the modified transaction size, 78 bytes.

If this sounds confusing, let's start again. The total value of the inputs in the transaction are 51531957, the value stored in the inChainInputValue variable. The current height as explained before is 427780 and the entryHeight is one less,

427779. The entryHeight is the temporary block height when the transaction entered the Bitcoins mempool. The value -1 is hard-coded.

The concept of a mempool is an area of memory where transactions get sent to by peers/miners. Miners constantly broadcast transactions that they receive to all and sundry.

When our peer received this transaction, the current or latest block mined was block number 427779, once again only when we ran our code. These transactions are still to be included in a block. No one knows how long they will stay in our mempool i.e. if any miner would ever include these transactions in the blockchain. A miner may include this transaction but that miner may be upstaged by another who found the block hash before him/her. When a transaction is finally included in the blockchain, all peers remove it from the mempool.

Thus, longer the transaction stays in the mempool, i.e. the longer the wait, the value of deltaPriority increases and the question is: do miners want to include such older transactions in the blockchain? Does this allow older transactions to get more traction? The older transactions must be included into the blockchain first, if we are being fair to seniors.

The value of deltaPriority is calculated by multiplying the total number of inputs by value with the difference in block height and entry height, $51531957 * (427780 - 427779)$ which is = to 51531957. This is the value of the old inChainInputValue. This value is then divided by the modified transaction size 78, $51531957 / 78$, and the answer is 660666.115385. Finally, the two priorities are added, $660666.115385 + 22867676.653846 = 23528342.769$ The result is the final priority of the transaction.

This program has many problems. The first is that the transaction is still in the mempool and not in a block saved in the blockchain. These are unconfirmed transactions. Therefore, an exception is thrown when there is access to the blockhash field of the data returned by the getrawtransaction command, as there is no such field present as of now. Also, there must be a better way to obtain the block height.

The program works on the premise that the transaction will be present in the mempool for only one block, but transactions can remain forever in the mempool if the fee is too low and then also be discarded all peers. This puts the calculation of the deltaPriority off track as the assumption made for a block height difference is only 1.

Since we had access to the source code, we learnt that the priority of the transaction is calculated by a function aptly called GetPriority. This fact saved plenty of time and eliminated writing printf's all over the Bitcoin source code.

Verifying the Transaction Priority that We Calculate

```
ch2913.py
import subprocess
import json
import time
def tdetails(txid , oindex):
    raw = subprocess.check_output(["bitcoin-cli" ,"getrawtransaction" , txid , "1" ])
    #print raw
    raw = json.loads(raw)
    btc = raw["vout"][oindex]["value"]
    btc = int(btc * 100000000)
    try:
        blockhash = raw["blockhash"]
    except:
        return (None , None)
    raw1 = subprocess.check_output(["bitcoin-cli" ,"getblock" , blockhash ])
```

```
raw1 = json.loads(raw1)
bheight = raw1["height"]
return (btc,bheight)

# if raw00['fee'] > 0.0040383:
# #print key1
# #print raw00
# break

raw0 = subprocess.check_output(["bitcoin-cli","getrawmempool","true"])
raw0 = json.loads(raw0)
for txid, raw00 in raw0.items():
    bheight = raw00["height"]
    startpriority = raw00["startingpriority"]
    currentpriority = raw00["currentpriority"]
    tsize = raw00["size"]
    tfee = raw00["fee"]
    time1 = raw00["time"]
    #print "Transaction Size %d fee %f height %d %s" % (tsize, tfee, bheight,
    time.ctime(time1))
    raw = subprocess.check_output(["bitcoin-cli","getrawtransaction",txid,"1"])
    #print raw
    raw = json.loads(raw)
    dResult = 0
    inChainInputValue = 0
    nTxSize = tsize
    for i in range(0,len(raw["vin"])):
        txid = raw["vin"][i]["txid"]
        vout = raw["vin"][i]["vout"]
        #print "(%d) Transaction Hash %s:%d" % (i, txid, vout)
        (btc,bheightt) = tdetails(txid, vout)
        if btc is None:
            continue
        diff = bheight - bheightt
        #print "height %d heightt %d diff %d" % (bheight, bheightt, bheight - bheightt)
        inChainInputValue = inChainInputValue + btc
        #print "Output Bitcoins %d Block height %d diff %d" % (btc, bheightt, diff)
        dResult = dResult + btc * diff
        #print "dResult %d inChainInputValue %d" % (dResult, inChainInputValue)
        scriptSig = raw["vin"][0]["scriptSig"]['hex']
        length = (len(scriptSig) ) / 2
        offset = 41 + min(110, length)
        #print "Length of scriptSig %d offset %d nTxSize %d" % (length, offset, nTxSize)
        if nTxSize > offset :
            nTxSize = nTxSize - offset
        #print "nTxSize %d" % nTxSize
    #print
```

```

entryPriority = 1.0 * dResult / nTxSize
deltaPriority = (-1.0 * inChainInputValue) / nTxSize
#print "entryPriority%f deltapriority%f" % (entryPriority , deltaPriority)
#totalpriority = entryPriority + deltaPriority
#print "TotalPriority%f" % totalpriority
#print "startingpriority%f currentPriority%f" % (startpriority , currentpriority)
if entryPriority - startpriority <= 0.0001:
    print "Transaction id %s: Ok" % txid
else:
    print "%s Not Ok entry %f starting %f Difference %f" % (txid , entryPriority , s
    tartpriority , entryPriority - startpriority)

```

Output

```

Transaction id
21ffc5e917d79b595c1906153512f550f505932c90777d26d4e663b28401f3c4: Ok
Transaction id
72eac18d831cebd2af187c3e2849ebda8d61c8048c63b32294e6a097da38d0b2: Ok
Transaction id
914a4d6029050265260a5813521b5d922f7ea971ef95786b3288257b26fe2c0f: Ok
e9815d2b46c92d16c6cc20ab6d2ac94c4c1d14f7beccb1e51f88571910cb6803 Not Ok
entry 41511536.936937 starting 41140898.214286 Difference 370638.722651
Transaction id
d7664e069846be640c6c17a1a453395c01a5c38fad2fc4ed59bf974755c708e4: Ok

```

The mempool is not a static entity, it keeps increasing and decreasing in size. Transactions come and go. Our mempool is very different from yours.

The `getrawmempool` command returns a dictionary where the key is an unconfirmed transaction in the mempool but these transactions have passed all the tests of time or those laid out by Bitcoin Core. The values in the dictionary are the transactions details like starting priority and fee and size etc. and much more. A loop iterates through each of these transactions present in the mempool. The six fields obtained are height, startingpriority, currentpriority, size, fee and time.

This transaction is available in the `getrawtransaction` command but the `blockhash` field will remain non-existent, as this transaction is in the mempool and not in the blockchain sitting in a confirmed block. Also, fields like the number of confirmations, time, etc. are simply not shown at the bottom of the output. Thus, it is safe to conclude is that the `getrawtransaction` command knows that this transaction exists only in the mempool.

Every input found in the mempool transaction is scanned. First requirement is the output that this input points to. So, always obtain the transaction hash value and the output index. The function `tdetails` returns the bitcoins that the output owns and the height of the transaction in the block. The hiccup here is that this transaction that owns Bitcoins may be in the mempool and not in the blockchain.

For instance, I send out a transaction paying my Bitcoin address some BTC's and immediately used that Bitcoin address to send some Bitcoins to another person. Both transactions are in the mempool and not in the blockchain. It takes a minimum of 10 minutes for the transactions to be spent. This happens again when some change is received in a transaction and we spend that change immediately. Both transactions are in the mempool. As a result, an exception is thrown when accessing the field `blockhash` in the function `tdetails`. Nevertheless, the offending output is still in the mempool. A value of `None` is returned for both, Bitcoins and height.

When the function `tdetails` returns a `None`, we simply go back to the start of the inner for loop.

The `diff` variable is used to age the inputs by subtracting the original entry height with the offset where the output live in the blockchain. Therefore, the bitcoins are multiplied with `diff` and the variable `dResult` is a running total of the input age. The `inChainInputValue` like before, is the running total of only the inputs.

The entry priority is the `dResult` divided by the modified size, `nTxSize`. This entry priority is checked with the starting priority obtained from the mempool, as they are floating point numbers. It is not a simple equality check.

A lot more is left with this program. For example, we wanted to remove the error we get but for some reason did not want to take a last look at the source code. Deadline calling. Not our finest hour. Also, when you have time try and figure out whether we have calculated the `deltaPriority` rightly or not. It has not been validated.

After all, if you do not calculate the fee properly, the heavens will not fall. Time agrees with us because in version 01.5, priority has been banished to segwit2x and to Bitcoin Cash and to Bitcoin Gold if it ever arrives.

CHAPTER 30

Building the Bitcoin Source Code

This chapter is our first chapter on our odyssey with the Bitcoin source code. Please follow our instructions to the T. Download the source code of Bitcoin using the web address <https://github.com/bitcoin/bitcoin>. The file downloaded is a zip file, on extracting the files through the unzip command, a folder bitcoin-master is created and it contains the Bitcoin source code.

We could have also use the git clone command but we preferred the direct zip download button.

The C++ code must be compiled first. The build command will finally give us the most important files, bitcoind and bitcoin-cli.

```
$. /autogen.sh
```

The first command we run is not something special as it is standard Linux folklore.

```
$. /configure --disable-tests --disable-bench --without-gui
```

The next command creates a Makefile which determines the .cpp files to be compiled. Plus, it tweaks the behavior of the linker.

The command configure is a standard Linux command and it checks if all software is present on your computer to build this program. With the configure command, we can specify the parts of the Bitcoin source we want to build. The prerequisites of building the source is very well documented.

On the Mac OS, we simply run one brew command to make sure that we have all the moving parts to build the Bitcoin source.

We disable tests, benchmarks and the GUI program. We have always used bitcoind from the command line. Plus, when we tried to build the GUI, we got Qt version issues.

The Bitcoin core version is present in the file validation.h for version 0.13 onwards and in older versions, main.h. We open the file and just change one line only.

The original line of code

```
static const unsigned int MAX_BLOCKFILE_SIZE = 0x8000000; // 128 MiB
```

The changed code

```
static const unsigned int MAX_BLOCKFILE_SIZE = 0x80000; // 128 MiB
```

Two zeroes are removed from the end of the variable MAX_BLOCKFILE_SIZE. The value of the variable is now a smaller value. In C++, a variable in all capital means it will never ever change its value, it's a constant. This is what the const keyword indicates. Every language has its idiosyncrasies.

```
$make -j6
```

The make command runs the compiler and the linker. The -j option is to specify the number of concurrent processes to run, we want 6 to speed things up.

There is no Bitcoin folder as we are starting clean.

Run the following bitcoin programs created in the src folder.

```
$. /src/bitcoind --printtoconsole
```

A screen full of some messages starts rolling out and a wallet gets created etc. The block folder gets filled up as usual with files. Given below is a listing after a few blocks get downloaded.

```
$ls -l
-rw---- 1 vijaymukhi staff 524083 Jan 17 12:24 blk00000.dat
-rw---- 1 vijaymukhi staff 524233 Jan 17 12:25 blk00001.dat
-rw---- 1 vijaymukhi staff 524239 Jan 17 12:26 blk00002.dat
-rw---- 1 vijaymukhi staff 524232 Jan 17 12:27 blk00003.dat
-rw---- 1 vijaymukhi staff 524146 Jan 17 12:28 blk00004.dat
-rw---- 1 vijaymukhi staff 524278 drwx---- 36 vijaymukhi staff 1224 Jan 17 12:31 index
-rw---- 1 vijaymukhi staff 97193 Jan 17 12:24 rev00000.dat
-rw---- 1 vijaymukhi staff 97721 Jan 17 12:25 rev00001.dat
-rw---- 1 vijaymukhi staff 98530 Jan 17 12:26 rev00002.dat
-rw---- 1 vijaymukhi staff 98339 Jan 17 12:27 rev00003.dat
-rw---- 1 vijaymukhi staff 97004 Jan 17 12:28 rev00004.dat
-rw---- 1 vijaymukhi staff 99527
```

For the first four blocks, the blk.dat files have a file size up to 512K B. Earlier our file size was capped at 128MB.

This happens because of a single line in the file validation.cpp or main.cpp. The offending lines are:

```
while (vinfoBlockFile[nFile].nSize + nAddSize >= MAX_BLOCKFILE_SIZE) {
    nFile++;
    if (vinfoBlockFile.size() <= nFile) {
        vinfoBlockFile.resize(nFile + 1);
    }
}
```

The code checks if the file size is larger than the constant MAX_BLOCKFILE_SIZE (128 MB).

We stop the bitcoind server gracefully by executing the 'bitcoin-cli stop' command or we press Ctrl-C in the window that runs bitcoind. This is why we never run the bitcoin server as a daemon. Killing it causes a reindexing of files and then years are lost simply waiting for the reindexing to end.

Now we use a different approach. We run the bitcoind sever that we downloaded a long time ago.

```
$bitcoind -printtoconsole
```

This program is located on our Mac in the folder /usr/local/bin/. Please note, this is not the server we just built.

We thought all hell would break loose, but nothing happened as such. Untouched by us, Bitcoin server kept downloading blocks and now the file size changes.

```
$ls -alSh
total 81048
-rw---- 1 vijaymukhi staff 96M Jan 17 12:47 blk00006.dat
-rw---- 1 vijaymukhi staff 13M Jan 17 12:47 rev00006.dat
```

The file size of the file blk0006.dat is already 96MB large. It simply proves that we can mix and match file sizes at our will. Please do not try this at home, we always wanted to say these lines in our book.

The condition in the source code is to check the file size. Whenever it is larger than the constant `MAX_BLOCKFILE_SIZE`, it creates a fresh empty blk file increasing the file number by 1 using the variable `nFile`. As a result, we now have two different sets of file sizes for the blk files.

Bitcoin code completely trusts the blockchain data on our disk.

The `grep` command finds out all the source files that have the word `getblockhash`.

```
$grep -rn . -e getblockhash
./rpc/blockchain.cpp:606:UniValue getblockhash(const JSONRPCRequest& request)
```

Your mileage may vary depending upon the Bitcoin Core version you use. We use `grep` at least 10 times a day to find stuff in files spread over multiple folders. Open the file `blockchain.cpp` in your favorite C++ editor. Yes, we have a favorite editor for every programming language we use. Then in the `getblockhash` function, add Vijay Mukhi Returns at the beginning of the string, as in.

```
UniValue getblockhash(const JSONRPCRequest& request)
{
    if (request.fHelp || request.params.size() != 1)
        throw runtime_error(
            "getblockhash index\n"
            "\nVijay Mukhi Returns hash of block in best-block-chain at index\n"
            "provided.\n"
            "\nArguments:\n"
```

Save the file and rebuild the bitcoin server. Then run the bitcoin server as always. No errors reported.

Now run the Bitcoin client in the `src` folder

```
$. /bitcoin-cli getblockhash
```

Output

```
error code: -1
error message:
$getblockhash index
```

```
Vijay Mukhi Returns hash of block in best-block-chain at index provided.
```

HAHAHA see my name.

When we run the original Bitcoin client.

```
$bitcoin-cli getblockhash
```

Output

```
error code: -1
error message:
getblockhash index
```

Vijay Mukhi Returns hash of block in best-block-chain at index provided.

The output remains the same as the code is changed in the Bitcoin server and not the Bitcoin client.

```
$ls -l bitcoind
-rwxr-xr-x 1 vijaymukhi staff 7141976 Jan 17 13:32 bitcoind

$ls -l bitcoin-cli
-rwxr-xr-x 1 vijaymukhi staff 611024 Jan 17 13:25 bitcoin-cli
```

All these commands are executed from the src folder. The date-time stamp of bitcoin server changes but for the bitcoin client, it remains the same. The bitcoin server has a newer date time stamp.

Now let's change the genesis block. The function called CMainParams in the chainparams.cpp file is reviewed. The numbers 123456789 is added to the first parameter.

```
genesis = CreateGenesisBlock(1231006505 + 123456789, 2083236893, 0x1d00ffff, 1,
50 * COIN);
```

This addition changes the time of creation of the genesis block.

Build build and run bitcoind.

```
$. /src/bitcoind -printtconsole
Assertion failed: (consensus.hashGenesisBlock ==
uint256S("0x00000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce2
6f")), function CMainParams, file chainparams.cpp, line 116.
Abort trap: 6
```

Bitcoin does not permit this change and hence it gives the above error. The genesis block is too close to Bitcoin's soul for anyone to mess with it. We are told that the error is simply on an assert function call, two lines later. So, we comment out this line code.

```
//assert(consensus.hashGenesisBlock ==
uint256S("0x00000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f"));
```

The comment characters in all programming languages are different. The UN once and for all, must standardize the comment characters as a # or a // or anything.

Output

```
2017-01-17 08:10:09 ERROR: ReadBlockFromDisk: Errors in block header at
CBlockDiskPos(nFile=0, nPos=8)
2017-01-17 08:10:09 *** Failed to read block
2017-01-17 08:10:09 Error: Error: A fatal internal error occurred, see debug.log for
details
Error: Error: A fatal internal error occurred, see debug.log for details
2017-01-17 08:10:09 Failed to connect best blockFailed to open mempool file from
disk. Continuing anyway.
```


Again, an error which clearly indicates no fooling around with the genesis block. So, undo the damage and get back the original.

At present, there is no file called bitcoin.conf in the Bitcoin folder. In the same validation.cpp or main.cpp, there is a function called OpenBlockFile.

```
FILE* OpenBlockFile(const CDiskBlockPos &pos, bool fReadOnly) {
    return OpenDiskFile(pos, "vijay", fReadOnly);
}

[~/Library/Application Support/Bitcoin/blocks]$ ls -l
total 36032
drwx----- 7 vijaymukhi staff 238 Jan 17 17:48 index
-rw----- 1 vijaymukhi staff 97193 Jan 17 17:58 rev00000.dat
-rw----- 1 vijaymukhi staff 1048576 Jan 17 18:01 rev00001.dat
-rw----- 1 vijaymukhi staff 524083 Jan 17 17:58 vijay00000.dat
-rw----- 1 vijaymukhi staff 16777216 Jan 17 18:01 vijay00001.dat

$bitcoin-cli getblockhash 1
00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048

[~/Library/Application Support/Bitcoin/blocks]$ bitcoin-cli getblockhash 3000
000000004a81b9aa469b11649996ecb0a452c16d1181e72f9f980850a1c5ecce
```

The original code replaces vijay with the words blk. With this change, they start with vijay. Now we understand why all our files in the blocks folder starts with the words blk. The command getblockhash also returns no error. To get the same effect, delete the Bitcoin folder, make the change and then make the code. Then run and you will see what we see.

Now is the time to pay homage to the Bitcoin core code. Some code snippets are given that helped us understand the code written in the previous chapters.

The first file we read a million times was the file, coins.h. We read and read the comments multiple times.

```
* The nCode value consists of:
* - bit 0: IsCoinBase()
* - bit 1: vout[0] is not spent
* - bit 2: vout[1] is not spent
* - The higher bits encode N, the number of non-zero bytes in the following bitvector.
* - In case both bit 1 and bit 2 are unset, they encode N-1, as there must be at
*   least one non-spent output).
*
* Example: 0104835800816115944e077fe7c803cfa57f29b36bf87c1d358bb85e
* <><><-----><-----><----->
* | \           |           /
* version code   vout[1]   height
*
* - version = 1
* - code = 4 (vout[1] is not spent, and 0 non-zero bytes of bitvector follow)
* - unspentness bitvector: as 0 non-zero bytes follow, it has length 0
* - vout[1]: 835800816115944e077fe7c803cfa57f29b36bf87c1d35
```

```
*      * 8358: compact amount representation for 60000000000 (600 BTC)
*      * 00: special txout type pay-to-pubkey-hash
*      * 816115944e077fe7c803cfa57f29b36bf87c1d35: address uint160
* - height = 203998
```

There are only two examples that explain the UTXO set format. If it were not for these comments and the code following, it would have been extremely difficult to understand the UTXO set.

The file, serialize.h has another set of comments that explain how variable length integers are stored.

```
/**
 * Variable-length integers: bytes are a MSB base-128 encoding of the number.
 * The high bit in each byte signifies whether another digit follows. To make
 * sure the encoding is one-to-one, one is subtracted from all but the last digit.
 * Thus, the byte sequence a[] with length len, where all but the last byte
 * has bit 128 set, encodes the number:
 *
 * (a[len-1] & 0x7F) + sum(i=1..len-1, 128^i*((a[len-i-1] & 0x7F)+1))
 *
 * Properties:
 * * Very small (0-127: 1 byte, 128-16511: 2 bytes, 16512-2113663: 3 bytes)
 * * Every integer has exactly one encoding
 * * Encoding does not depend on size of original integer type
 * * No redundancy: every (infinite) byte sequence corresponds to a list
 * of encoded integers.
 *
 * 0:      [0x00]      256:      [0x81 0x00]
 * 1:      [0x01]      16383:     [0xFE 0x7F]
 * 127:     [0x7F]      16384:     [0xFF 0x00]
 * 128:     [0x80 0x00] 16511:     [0x80 0xFF 0x7F]
 * 255:     [0x80 0x7F] 65535:     [0x82 0xFD 0x7F]
 * 2^32:     [0x8E 0xFE 0xFE 0xFF 0x00]
 */
```

Invaluable to us. The chData variable and the related code in our programs comes from the code in the serialize.h file.

```
template<typename Stream, typename I>
I ReadVarInt(Stream& is)
{
    I n = 0;
    while(true) {
        unsigned char chData = ser_readdata8(is);
        n = (n << 7) | (chData & 0x7F);
        if (chData & 0x80)
            n++;
        else
            return n;
    }
}
```

The client version is nothing but a macro or a #define.

```
#define CLIENT_VERSION_MAJOR 0
#define CLIENT_VERSION_MINOR 13
#define CLIENT_VERSION_REVISION 99
#define CLIENT_VERSION_BUILD 0

static const int CLIENT_VERSION =
    1000000 * CLIENT_VERSION_MAJOR
    + 10000 * CLIENT_VERSION_MINOR
    + 100 * CLIENT_VERSION_REVISION
    + 1 * CLIENT_VERSION_BUILD;
```

The client version is calculated in this manner, especially for wallets. As you see, this is version 13 and not version 15.

For index folder programming, we learnt the following from the source.

The Bitcoin source has a file called chain.h that contains a class called CBlockIndex. The members of this class get written to disk. The class makes it very clear that not all members get written to disk, some are for memory only.

Hey! Wait a minute! The order of variables in the class CBlockIndex does not match the order for the values stored in the index! Or do they, how do we know.

There is a class called CDiskBlockIndex that derives from the class CBlockIndex. There is a method called SerializationOp which writes the index folder data to disk.

```
inline void SerializationOp(Stream& s, Operation ser_action) {
    int nVersion = s.GetVersion();
    if (!s.GetType() & SER_GETHASH)
        READWRITE(VARINT(nVersion));

    READWRITE(VARINT(nHeight));
    READWRITE(VARINT(nStatus));
    READWRITE(VARINT(nTx));
    if (nStatus & (BLOCK_HAVE_DATA | BLOCK_HAVE_UNDO))
        READWRITE(VARINT(nFile));
    if (nStatus & BLOCK_HAVE_DATA)
        READWRITE(VARINT(nDataPos));
    if (nStatus & BLOCK_HAVE_UNDO)
        READWRITE(VARINT(nUndoPos));

    // block header
    READWRITE(this->nVersion);
    READWRITE(hashPrev);
    READWRITE(hashMerkleRoot);
    READWRITE(nTime);
    READWRITE(nBits);
    READWRITE(nNonce);
}
```

This code decides on the data that gets written to disk and in what order. It is here that we see the block header fields being written in the same order. It is a simple way to understand the structure of the index folder. There is no reason to

understand the C code, you do not have to be a C nerd. This is the easiest way to sound intelligent as not many people would have read the source of the Bitcoin core. Bitcoin is not written in C++ but a very advanced version of C++.

In the same file chain.h, there is an enum called BlockStatus that has valid values of field nStatus.

```
enum BlockStatus: uint32_t {
    //! Unused.
    BLOCK_VALID_UNKNOWN = 0,
    //! Parsed, version ok, hash satisfies claimed PoW, 1 <= vtx count <= max, timestamp not in future
    BLOCK_VALID_HEADER = 1,
    //! All parent headers found, difficulty matches, timestamp >= median previous, checkpoint.
    Implies all parents
    //! are also at least TREE.
    BLOCK_VALID_TREE = 2,
    /**
     * Only first tx is coinbase, 2 <= coinbase input script length <= 100, transactions valid, no duplicate txids,
     * sigops, size, merkle root. Implies all parents are at least TREE but not necessarily TRANSACTIONS. When
     * all
     * parent blocks also have TRANSACTIONS, CBlockIndex::nChainTx will be set.
     */
    BLOCK_VALID_TRANSACTIONS = 3,
    //! Outputs do not overspend inputs, no double spends, coinbase output ok, no immature
    coinbase spends, BIP30.
    //! Implies all parents are also at least CHAIN.
    BLOCK_VALID_CHAIN = 4,
    //! Scripts & signatures ok. Implies all parents are also at least SCRIPTS.
    BLOCK_VALID_SCRIPTS = 5,
    //! All validity bits.
    BLOCK_VALID_MASK = BLOCK_VALID_HEADER | BLOCK_VALID_TREE | BLOCK_VALID_TRANSACTIONS |
    BLOCK_VALID_CHAIN | BLOCK_VALID_SCRIPTS,
    BLOCK_HAVE_DATA = 8, //!< full block available in blk*.dat
    BLOCK_HAVE_UNDO = 16, //!< undo data available in rev*.dat
    BLOCK_HAVE_MASK = BLOCK_HAVE_DATA | BLOCK_HAVE_UNDO,
    BLOCK_FAILED_VALID = 32, //!< stage after last reached validness failed
    BLOCK_FAILED_CHILD = 64, //!< descends from failed block
    BLOCK_FAILED_MASK = BLOCK_FAILED_VALID | BLOCK_FAILED_CHILD,
    BLOCK_OPT_WITNESS = 128, //!< block data in blk*.data was received with a witness-enforcing client
};
```

Now you can see how we became experts at understanding a block status type field.

The file compressor.h shows three special cases of hash types.

```
/** Compact serializer for scripts.
 *
 * It detects common cases and encodes them much more efficiently.
```

```

* 3 special cases are defined:
* * Pay to pubkey hash (encoded as 21 bytes)
* * Pay to script hash (encoded as 21 bytes)
* * Pay to pubkey starting with 0x02, 0x03 or 0x04 (encoded as 33 bytes)
*
* Other scripts up to 121 bytes require 1 byte + script length. Above
* that, scripts up to 16505 bytes require 2 bytes + script length.
*/

```

Now to the next thing. The icing on the cake. The problem with understanding bitcoin is its half a million blocks and corresponding rev files. Let's not forget the two folders, chainstate and index which are many GB's large.

It is extremely difficult to work with such large data sets? So, we first make a copy of the Bitcoin folder, take a backup. Then we open the file main.cpp or validation.cpp. Now we locate a function called LoadExternal file and add some code there.

```

try {
    // read block
    uint64_t nBlockPos = blkdat.GetPos();
    if (dbp)
        dbp->nPos = nBlockPos;
    blkdat.SetLimit(nBlockPos + nSize);
    blkdat.SetPos(nBlockPos);
    CBlock block;
    blkdat >> block;
    /*
    if (g_blockno >= 10)
    {
        FlushStateToDisk();
        printf("Quitting Out for good\n\n");
        exit(0);
    }
    */
    nRewind = blkdat.GetPos();
    //printf("\nmain.cpp LoadExternalBlockFile Start Block Number is %d \n", g_blockno);
    // detect out of order blocks, and store them for later
    uint256 hash = block.GetHash();

```

A global variable called g_blockno is created by us. The variable increases by 1 each time we loop through a block. The minute 10 blocks are accessed, we quit out. This is how we understood concepts like a rev or undo file.

Bitcoin stores lots of data into its memory buffers. So, instead of patiently waiting, call a Bitcoin Core function, FlushStateToDisk is called. This function writes all the Bitcoin entities like the UTXO set, the index folder etc to disk. Then and then quit out. If we do not call this function, nothing gets written to disk.

We now have with us the state of Bitcoin after 10 blocks. It is much easier to understand the workings of Bitcoin now. When we finish, we simply copy the original Bitcoin folder over. Actually, we rename the folder as copying 200GB takes too long.

Once again, we pay homage to the Bitcoin Core developers for creating a crypto- currency that has stood the test of time.

Let's end with an error. Something we fail to understand. When we build, and run the Bitcoin source on our Retina iMac and the Mac Pro, we get an error as displayed below. This error does not come while running any other version of the bitcoind executable.

```
2017-01-17 18:31:28 libevent: kq_init: detected broken kqueue; not using.:
Undefined error: 0

2017-01-17 18:31:28 CDBEnv::Open: LogDir=/Users/vijaymukhi/Library/Application
Support/Bitcoin/testnet3/database
ErrorFile=/Users/vijaymukhi/Library/Application Support/Bitcoin/testnet3/db.log
DB_ENV->set_lk_detect: unknown deadlock detection mode specified
DB_ENV->stat_print: method not permitted before handle's open method
2017-01-17 18:31:28 ERROR: CDBEnv::Open: Error 22 opening database
environment: Invalid argument

2017-01-17 18:31:28 Moved old /Users/vijaymukhi/Library/Application
Support/Bitcoin/testnet3/database to /Users/vijaymukhi/Library/Application
Support/Bitcoin/testnet3/database.1484677888.bak. Retrying.
2017-01-17 18:31:28 CDBEnv::Open: LogDir=/Users/vijaymukhi/Library/Application
Support/Bitcoin/testnet3/database ErrorFile=/Users/vijaymukhi/Library/Application
Support/Bitcoin/testnet3/db.log
DB_ENV->set_lk_detect: unknown deadlock detection mode specified
DB_ENV->stat_print: method not permitted before handle's open method
2017-01-17 18:31:28 ERROR: CDBEnv::Open: Error 22 opening database
environment: Invalid argument

2017-01-17 18:31:28 Error: Error initializing wallet database environment
"/Users/vijaymukhi/Library/Application Support/Bitcoin/testnet3"!
Error: Error initializing wallet database environment
"/Users/vijaymukhi/Library/Application Support/Bitcoin/testnet3"!
2017-01-17 18:31:28 Shutdown: In progress...
2017-01-17 18:31:28 scheduler thread interrupt
2017-01-17 18:31:28 StopNode()
2017-01-17 18:31:28 Shutdown: done
```

The only good news is that the server shuts down by itself. No idea what we installed on these machines that all hell broke loose. Moral of the story. Always experiment on a machine that has no software installed. Or take a backup of all the files, in case there is a server crash. Just what we did, but unintentionally.

Not for the last time, this book would not have written if we did not have access to the Bitcoin source code. We bet our last penny on Bitcoin Core as Bitcoin Cash or segwit2x do not have the support of the core developers, they will not be able to take the Bitcoin ecosystem into the 23rd century.

We pay homage to the Bitcoin source code and the people who wrote it.

CHAPTER 31

Testing Bitcoin for Bugs

The Bitcoin developers test their software using two languages, Python and C++.

As a simple revision, when we run the shell script `autogen.sh`, a file called `makefile.in` is created. This file explicitly states that it has been created using `automake` with the file `Makefile.am`. One more file called `configure` is created at the same time and its creator is `autoconf`. This small `autogen` script file on execution, categorically calls `automake` and `autoconf` internally and creates two files at the bare minimum.

As we said earlier, this `configure` script checks for all the required files on disk and then compiles and builds our code. It looks out for files like `brew`, `java`, `boost` etc. If these files are not installed, the `configure` script fails and the `Makefile` file is not created.

The only reason why we are repeating all this is because you need to run the `configure` script without the `—disable-tests` option.

The `make` command is the heart and soul of the build process. People have written voluminous books on the `make` program and still find it abridged and unfinished.

The `make` program looks for a file called `Makefile` in the current folder. This file, `Makefile` contains all the relevant commands with their command line options for the compiler and the linker. It also picks up other `.cpp` and relevant files that are to be compiled and linked to generate an executable file. No `Makefile`, no `bitcoind` executable.

The `-j` option ensures all the cores on our computer are used. Miss this option and a tea break becomes a 5-star lunch break.

To sum up, script `autogen.sh` produces two files, `makefile.in` and `configure`. Then, the `configure` command creates a file, `Makefile` and finally running the `make` command generates the final executable programs, `bitcoind` and `bitcoin-cli`.

The code for testing is present in the `src/test` folder. We first open a file, `Makefile.test.include` in the `src` folder.

```
/Users/vijaymukhi/Dropbox/bitcoin/src/Makefile.test.include
```

Jump to line 138 and add a filename called `mukhi.cpp`. With version 0.15 we add this file name to line number

```
test/versionbits_tests.cpp \  
test/uint256_tests.cpp \  
test/univalue_tests.cpp \  
test/mukhi.cpp \  
test/util_tests.cpp
```

The `\` refers to a line in continuation.

Now move to the folder `src/test` and run the `make` command again.

```
make -j6
```

Output

```
CXX test/test_test_bitcoin-mukhi.o
clang: error: no such file or directory: './test/mukhi.cpp'
clang : error: no input files
```

The make command throws an error indicating that there is no file called mukhi.cpp. So, create an empty file called mukhi.cpp in the test directory and run the make command again.

```
make -j6
```

Output

```
/Library/Developer/CommandLineTools/usr/bin/make -C .. bitcoin_test
CXX test/test_test_bitcoin-mukhi.o
CXXLD test/test_bitcoin
```

Haha, no complaints at all.

```
Makefile.tests.include
    test/univalue_tests.cpp \
    test/mukhi_tests.cpp \
    test/util_tests.cpp
```

Now, rename the file, mukhi.cpp to mukhi_tests.cpp. Every test file in the bitcoin test ecosystem ends with _tests. Maybe a good luck charm!

```
z
/usr/bin/osascript -e 'tell application "System Events" to tell process "Terminal" to
keystroke "k" using command down'
make -j6
if [ $? -eq 0 ]
then
./test_bitcoin --log_level=all --run_test=vijay
else
echo "Vijay Mukhi is an embecile"
fi
```

We have spent over 35 years writing code in C. So, C will always be our first, last and only love when it comes to programming languages. Please do not mistaken C++ as a better C.

A batch file or a shell script is used to automate most redundant tasks. Our scripts are always called z or a as they will then be shown at the beginning or end in the ls command or dir listing. We make our script file, z executable by running the chmod command, as shown below. We make no assumption on what you know or do not know.

```
chmod 777 z
```

The first command uses a very obscure looking mac os x script command and blanks out the screen. This command clears not only the screen but also the screen buffer. A better clear command.

The make command is called in next. Using the if statement from the bash scripting language, the return value of the last command is checked to be a 0. The funny looking symbols \$? evaluates how the last command exited. If the last command ended successfully, the test script is executed otherwise an error message is displayed.

The best way to remember an `endif` is `fi`, the reverse of an `if`.

The `make` command simply builds an executable called `test_bitcoin` in the `src/test` folder. The option `log_level` allows the program, `bitcoin_test` to be a lot more verbose. The `run_test` option specifies the tests to be executed. There are command line options to run all the tests, but our concern is to only run a test called `vijay`.

The file `mukhi_tests.cpp` is empty at present. The build file `bitcoin_test` has no test named `vijay`, hence the error. The error message is shown below.

Output

```
/Library/Developer/CommandLineTools/usr/bin/make -C .. bitcoin_test
CXX test/test_test_bitcoin-mukhi_tests.o
CXXLD test/test_bitcoin
Test setup error: no test cases matching filter or all test cases were disabled
```

Creating a Test Called Vijay

```
ch3101.cpp
#include "test_bitcoin.h"
#include <boost/test/unit_test.hpp>
BOOST_FIXTURE_TEST_SUITE(vijay, BasicTestingSetup)
BOOST_AUTO_TEST_CASE(mukhi)
{
}
BOOST_AUTO_TEST_SUITE_END()
```

Output

```
Running 1 test case...
Entering test module "Bitcoin Test"
test/mukhi_tests.cpp:3: Entering test suite "vijay"
test/mukhi_tests.cpp:4: Entering test case "mukhi"
Test case vijay/mukhi did not check any assertions
test/mukhi_tests.cpp:4: Leaving test case "mukhi"; testing time: 16375us
test/mukhi_tests.cpp:3: Leaving test suite "vijay"; testing time: 16404us
Leaving test module "Bitcoin Test"; testing time: 16442us
*** No errors detected
```

The above program has only the bare minimum code required to create a test called `vijay`.

A fair warning. Our program file is named as `ch3101.cpp`, but you must name it or rename it to `mukhi_tests.cpp`.

The program does nothing. It is created to ensure that this file, `mukhi_tests.cpp` which was added to the `makefile`, is compiled and its compiled code is placed in the file `bitcoin_test`.

In C/C++, header files are a necessary evil as they bring in entities like macros or `#defines`. They, at times, also bring in actual code. The boost library code is included in the file using the `unit_test.hpp` header file. This library has code for error checks. The rules to follow when writing boost error checks are hidden from our view due to macros (which are in caps).

An entity called suite contains all the test cases so it is created first.

Our suite called vijay is based on the predefined boost magic suite, BasicTestingSetup. This suite name is also given to the option, run_test. The macro used is called BOOST_FIXTURE_TEST_SUITE.

Hence, no errors this time.

The macro SUITE_END simply signals the end of all tests.

A test is created in between these macros using the macro TEST_CASE. Our one and only test is called mukhi.

Let's look at our output. There is only one test case, this explains the 1 in the running output display. We have no idea why the test module is called Bitcoin Test.

The output states that on line number 3, there is a suite called vijay. At line number 4, is the test case, mukhi. In the test case mukhi, there is no code that checks for any condition, hence the warning message. Then, the test mukhi and suite vijay exit at the same above line numbers. However, an extra time element kicks in, as in the time taken for the test to run? In the end, a message declares no errors detected.

When the log_level is off, only the last line is displayed.

It is one of the simplest unit tests written using the boost library and forms the base over which all other tests are written.

The tests written by the Bitcoin developers are now explained one line a time.

```
test_bitcoin.cpp
#define BOOST_TEST_MODULE Sonal Test
#include "test_bitcoin.h"
```

Output

```
Entering test module "Sonal Test"
Leaving test module "Sonal Test"; testing time: 13513us
```

In the field test_bitcoin.cpp, the first line is changed to add a name, Sonal. This file is present in the src/test folder.

Running Two Test Cases.

```
ch3102.cpp
#include "test_bitcoin.h"
#include <boost/test/unit_test.hpp>
BOOST_FIXTURE_TEST_SUITE(vijay, BasicTestingSetup)
BOOST_AUTO_TEST_CASE(mukhi)
{
}
BOOST_AUTO_TEST_CASE(sonal)
{
}
BOOST_AUTO_TEST_SUITE_END()
```

Output

```
Running 2 test cases...
Entering test module "Sonal Test"
test/mukhi_tests.cpp:3: Entering test suite "vijay"
test/mukhi_tests.cpp:4: Entering test case "mukhi"
```

```

Test case vijay/mukhi did not check any assertions
test/mukhi_tests.cpp:4: Leaving test case "mukhi"; testing time: 13315us
test/mukhi_tests.cpp:7: Entering test case "sonal"
Test case vijay/sonal did not check any assertions
test/mukhi_tests.cpp:7: Leaving test case "sonal"; testing time: 882us
test/mukhi_tests.cpp:3: Leaving test suite "vijay"; testing time: 14247us
Leaving test module "Sonal Test"; testing time: 14285us

*** No errors detected

```

The suite is a higher-level entity. Here, there are two test cases, mukhi and sonal. The output shows 2 tests being executed. First entry is in test case mukhi and then comes the test case sonal.

Using the printf Function to Display the Output

```

ch3103.cpp
#include "test_bitcoin.h"
#include <boost/test/unit_test.hpp>
BOOST_FIXTURE_TEST_SUITE(vijay, BasicTestingSetup)
BOOST_AUTO_TEST_CASE(mukhi)
{
    printf("Hell\n");
}
BOOST_AUTO_TEST_SUITE_END()

```

Output

```

test/mukhi_tests.cpp:4: Entering test case "mukhi"
Hell
Test case vijay/mukhi did not check any assertions
test/mukhi_tests.cpp:4: Leaving test case "mukhi"; testing time: 13439us

```

The boost framework calls our test case mukhi and executes all the code in this so-called function. Any C++ code can be given here. It is a bad idea though to use the printf function as it blemishes the output. We need another function for logging.

Calling the Function glibc_sanity_test to Check Memory Accesses

```

ch3104.cpp
#include "test_bitcoin.h"
#include <boost/test/unit_test.hpp>
bool glibc_sanity_test();
BOOST_FIXTURE_TEST_SUITE(vijay, BasicTestingSetup)
BOOST_AUTO_TEST_CASE(mukhi)
{
    bool a = glibc_sanity_test() == true;
    printf("Hell %d\n", a);
}
BOOST_AUTO_TEST_SUITE_END()

```

Output

```
test/mukhi_tests.cpp:5: Entering test case "mukhi"
In glibc_sanity.cpp
Hell 1
Test case vijay/mukhi did not check any assertions
test/mukhi_tests.cpp:5: Leaving test case "mukhi"; testing time: 14081us
```

One of the many reasons we prefer using C over C++ is that C allows you to shoot yourself in your foot, if that is what you want. We are firmly in the C camp to ignore the C++ barbs. A function in C++ demands the function's prototype and directives of calling it, upfront. The C++ programming language insists on using the right parameters and parameter types when calling a function. Here, the prototype of function `glibc_sanity_test` is stated before calling it.

The return type of this function is checked. If it is true, the bool variable, `a` will have a value of 1, if false then `a` will be 0. The value of `a` is displayed accordingly.

In this example, we have used code written by the core Bitcoin developers to check if Bitcoin can copy to and from memory locations. It demonstrates how memory is handled by Bitcoin code. This is actual, real test that Bitcoin core performs.

Nothing stops us from writing/adding code in the Bitcoin source and calling it in our own test. The file that carries this function is found in the folder:

/Users/vijaymukhi/Dropbox/bitcoin/src/compat/

```
glibc_sanity.cpp
bool glibc_sanity_test()
{
    #if defined(HAVE_SYS_SELECT_H)
    if (!sanity_test_fdelt())
        return false;
    #endif
    printf("In glibc_sanity.cpp\n");
    return sanity_test_memcpy<1025>();
}
```

This code clarifies why the `printf` function is called. The `glibc_sanity_test` function simply calls a standard C library function called `memcpy`, and copies data from one memory location to another. Please add the `printf` to an actual Bitcoin Core test and feel the difference.

A macro displays the Output the boost way

```
ch3105.cpp
#include "test_bitcoin.h"
#include <boost/test/unit_test.hpp>
#include "compat/sanity.h"
BOOST_FIXTURE_TEST_SUITE(vijay, BasicTestingSetup)
BOOST_AUTO_TEST_CASE(mukhi)
{
    BOOST_CHECK_MESSAGE(glibc_sanity_test() == true, "Vijay Mukhi is a embecile");
}
```

```
BOOST_AUTO_TEST_SUITE_END()
```

Output

```
test/mukhi_tests.cpp:5: Entering test case "mukhi"
In glibc_sanity.cpp
test/mukhi_tests.cpp:7: info: check 'Vijay Mukhi is a embecile' has passed
test/mukhi_tests.cpp:5: Leaving test case "mukhi"; testing time: 15010us
```

Finally, a test that follows all the Bitcoin Core rules! The macro `BOOST_CHECK_MESSAGE` takes only two parameters, a logical condition and the message to be displayed.

The function prototype is defined in one of the header file. The warning message of the output disappears now, as now there is a condition in our test case. There are no memory copy issues here.

Failing a Bitcoin test

```
ch3106.cpp
#include "test_bitcoin.h"
#include <boost/test/unit_test.hpp>
#include "key.h"
BOOST_FIXTURE_TEST_SUITE(vijay, BasicTestingSetup)
BOOST_AUTO_TEST_CASE(mukhi)
{
    BOOST_CHECK_MESSAGE(ECC_InitSanityCheck() == true, "openssl ECC test");
}
BOOST_AUTO_TEST_SUITE_END()
```

Output

```
Running 1 test case...
Entering test module "Sonal Test"
test/mukhi_tests.cpp:4: Entering test suite "vijay"
test/mukhi_tests.cpp:5: Entering test case "mukhi"
In key.cpp
test/mukhi_tests.cpp:7: error: in "vijay/mukhi": openssl ECC test
test/mukhi_tests.cpp:5: Leaving test case "mukhi"; testing time: 13075us
test/mukhi_tests.cpp:4: Leaving test suite "vijay"; testing time: 13084us
Leaving test module "Sonal Test"; testing time: 13111us
*** 1 failure is detected in the test module "Sonal Test"
```

This test fails. We made sure that the function `ECC_InitSanityCheck` returns a false and hence it fails.

```
src folder
key.cpp
bool ECC_InitSanityCheck() {
    printf("In key.cpp\n");
    return false;
    CKey key;
    key.MakeNewKey(true);
    CPubKey pubkey = key.GetPubKey();
}
```

```
    return key.VerifyPubKey(pubkey);  
}
```

We short-circuited the function and simply returned false after adding our if statement. This needling is more to check if errors can be caught by the boost library test framework. Please undo the return false statement.

Making Sure that Our Test Fails Using the Macro BOOST_CHECK

```
ch3107.cpp  
#include "test_bitcoin.h"  
#include <boost/test/unit_test.hpp>  
#include "key.h"  
BOOST_FIXTURE_TEST_SUITE(vijay, BasicTestingSetup)  
BOOST_AUTO_TEST_CASE(mukhi)  
{  
    BOOST_CHECK(false);  
    printf("After\n");  
}  
BOOST_AUTO_TEST_SUITE_END()
```

Output

```
test/mukhi_tests.cpp:5: Entering test case "mukhi"  
test/mukhi_tests.cpp:7: error: in "vijay/mukhi": check false has failed  
After  
test/mukhi_tests.cpp:5: Leaving test case "mukhi"; testing time: 14136us  
test/mukhi_tests.cpp:4: Leaving test suite "vijay"; testing time: 14145us  
Leaving test module "Sonal Test"; testing time: 14174us  
*** 1 failure is detected in the test module "Sonal Test"
```

The macro BOOST_CHECK is simple as it takes only one parameter, a condition check. There is no message to display. A value of false is hard coded and hence an error occurs and an internal message is displayed. Nevertheless, boost does not stop processing our code. Therefore, the printf displays After, as shown in the output.

Checking if the Strings Follow the Encoding and Decoding Rules for base64

```
ch3108.cpp  
#include "test_bitcoin.h"  
#include <boost/test/unit_test.hpp>  
#include "utilstrencodings.h"  
BOOST_FIXTURE_TEST_SUITE(vijay, BasicTestingSetup)  
BOOST_AUTO_TEST_CASE(mukhi)  
{  
    std::string strEnc = EncodeBase64("foobar");  
    BOOST_CHECK(strEnc == "Zm9vYmFy");  
    std::string strDec = DecodeBase64(strEnc);  
    BOOST_CHECK(strDec == "foobar");  
}
```

```
BOOST_AUTO_TEST_SUITE_END()
```

Output

```
test/mukhi_tests.cpp:5: Entering test case "mukhi"
test/mukhi_tests.cpp:8: info: check strEnc == "Zm9vYmFy" has passed
test/mukhi_tests.cpp:10: info: check strDec == "foobar" has passed
test/mukhi_tests.cpp:5: Leaving test case "mukhi"; testing time: 15512us
```

Here, we have a practical test case taken from an original Bitcoin test to check the workings of the base64 encoding. The core developers of Bitcoin have kept in mind that we may want to build tests which check their code. Someone, somewhere wrote code in a function, `EncodeBase64` to encode a string into base64 and then another function, `DecodeBase64` to decode it back to the same string.

This function, `EncodeBase64` is given the string `foobar`. The base64 encoded string returned is `Zm9vYmFy`. This encoded string, `Zm9vYmFy` is given to the `DecodeBase64` function to check if the resultant string is the original one, `foobar`. The macro `BOOST_CHECK` will return an error if the original string `foobar` is not the same.

The original test code creates a vector of different strings and their base64 equivalents. They are created to test extreme cases of base64 encodings. Each minor and major version goes through these stringent error tests.

Checking the Code to Compress and Decompress Amounts Work

```
ch3109.cpp
#include "test_bitcoin.h"
#include <boost/test/unit_test.hpp>
#include <stdint.h>
BOOST_FIXTURE_TEST_SUITE(vijay, BasicTestingSetup)
bool static TestPair(uint64_t dec, uint64_t enc)
{
    printf("%llu\n", dec);
    return CTxOutCompressor::CompressAmount(dec) == enc &&
        CTxOutCompressor::DecompressAmount(enc) == dec;
}
BOOST_AUTO_TEST_CASE(mukhi)
{
    BOOST_CHECK(TestPair(COIN,0x9));
}
BOOST_AUTO_TEST_SUITE_END()
```

Output

```
test/mukhi_tests.cpp:12: Entering test case "mukhi"
100000000
test/mukhi_tests.cpp:14: info: check TestPair(COIN,0x9) has passed
test/mukhi_tests.cpp:12: Leaving test case "mukhi"; testing time: 14178us
```

This feels like home! We spent a fortune trying to understand how amounts are compressed in the bitcoin universe. The class `CTxOutCompressor` calls these two functions, `CompressAmount` and `DecompressAmount`.

The program calls the function `TestPair`, the name is kept in sync with the original test code. The original test also runs a series of tests on some zillion amounts to be compressed and decompressed. Ours is a small subset.

```
./test_bitcoin --log_level=all --run_test=compress_tests | wc -l
```

Output

```
*** No errors detected
```

The program test_bitcoin is executed with a test called compress_tests, which runs a little over half a million tests. The above code is taken from the test compress_tests.cpp.

A Transaction Goes Through Many Tests

```
ch3110.cpp
#include "test_bitcoin.h"
#include <boost/test/unit_test.hpp>
#include "consensus/validation.h"
#include "validation.h" // For CheckTransaction
#include "policy/policy.h"
inline bool MoneyRange1(const CAmount& nValue)
{
    return (nValue >= 0 && nValue <= MAX_MONEY);
}
bool CheckTransaction1(const CTransaction& tx, CValidationState &state, bool
fCheckDuplicateInputs)
{
    // Basic checks that don't depend on any context
    printf("Empty vin %d vout %lu\n", tx.vin.empty(), tx.vin.size());
    if (tx.vin.empty())
        return state.DoS(10, false, REJECT_INVALID, "bad-txns-vin-empty");
    if (tx.vout.empty())
        return state.DoS(10, false, REJECT_INVALID, "bad-txns-vout-empty");
    // Size limits (this doesn't take the witness into account, as that hasn't been
    checked for malleability)
    const unsigned int size = ::GetSerializeSize(tx, SER_NETWORK, PROTOCOL_VERSION |
SERIALIZE_TRANSACTION_NO_WITNESS);
    printf("Size %d %d\n", size, MAX_BLOCK_BASE_SIZE);
    if (size > MAX_BLOCK_BASE_SIZE)
        return state.DoS(100, false, REJECT_INVALID, "bad-txns-oversize");
    // Check for negative or overflow output values
    CAmount nValueOut = 0;
    for (const auto& txout : tx.vout)
    {
        if (txout.nValue < 0)
            return state.DoS(100, false, REJECT_INVALID, "bad-txns-vout-negative");
        if (txout.nValue > MAX_MONEY)
            return state.DoS(100, false, REJECT_INVALID, "bad-txns-vout-toolarge");
        nValueOut += txout.nValue;
        printf("Amount %lld:%lld\n", txout.nValue, MAX_MONEY);
        if (!MoneyRange1(nValueOut))
```



```

        return state.DoS(100, false, REJECT_INVALID, "bad-txns-txouttotal-toolarge");
    }
    // Check for duplicate inputs - note that this check is slow so we skip it in CheckBlock
    if (fCheckDuplicateInputs) {
        std::set<COutPoint> vInOutPoints;
        for (const auto& txin : tx.vin)
        {
            printf("PrevOut %s:%d\n", txin.prevout.hash.ToString().c_str(), txin.prevout.n);
            if (!vInOutPoints.insert(txin.prevout).second)
                return state.DoS(100, false, REJECT_INVALID, "bad-txns-inputs-duplicate");
        }
    }
    printf("Coinbase Transaction %d\n", tx.IsCoinBase());
    if (tx.IsCoinBase())
    {
        if (tx.vin[0].scriptSig.size() < 2 || tx.vin[0].scriptSig.size() > 100)
            return state.DoS(100, false, REJECT_INVALID, "bad-cb-length");
    }
    else
    {
        for (const auto& txin : tx.vin)
        {
            printf("prevout %s:%d\n", txin.prevout.ToString().c_str(), txin.prevout.IsNull());
            if (txin.prevout.IsNull())
                return state.DoS(10, false, REJECT_INVALID, "bad-txns-prevout-null");
        }
    }
    return true;
}
BOOST_FIXTURE_TEST_SUITE(vijay, BasicTestingSetup)
BOOST_AUTO_TEST_CASE(mukhi)
{
    printf("\n");
    const char *transaction =
        "0100000001b14bdc3e01bdaad36cc08e81e69c82e1060bc14e518db2b49aa43ad90b"
        "a26000000000490047304402203f16c6f40162ab686621ef3000b04e75418a0c0cb2d8ae"
        "bea c894ae360ac1e780220ddc15ecd3507ac48e1681a33eb60996631bf6bf5bc0a0682"
        "c4db743ce7ca2b01ffffff0140420f000000000001976a914660d4ef3a743e3e696ad9903"
        "64e555c271ad504b88ac00000000";
    CDataStream stream(ParseHex(transaction), SER_NETWORK, PROTOCOL_VERSION);
    CTransaction tx(deserialize, stream); // replace deserialize with {}
    CValidationState state;
    BOOST_CHECK(CheckTransaction1(tx, state, true));
    BOOST_CHECK(state.IsValid());
    printf("\n");
}

```

```
BOOST_AUTO_TEST_SUITE_END()
```

Output

```
test/mukhi_tests.cpp:64: Entering test case "mukhi"
```

```
Empty vin 0 vout 1
```

```
Size 158 1000000
```

```
Amount 1000000:2100000000000000
```

```
PrevOut 60a20bd93aa49ab4b28d514ec10b06e1829ce6818ec06cd3aabd013ebcdc4bb1:0
```

```
Coinbase Transaction 0
```

```
prevout COutPoint(60a20bd93a, 0):0
```

```
test/mukhi_tests.cpp:71: info: check CheckTransaction1(tx, state, true) has passed
```

```
test/mukhi_tests.cpp:72: info: check state.IsValid() has passed
```

```
test/mukhi_tests.cpp:64: Leaving test case "mukhi"; testing time: 14463us
```

It is assumed that Bitcoin transactions follow all the zillion rules laid out in the bips. The above program verifies some of these tests and claims. FYI, there are functions in the Bitcoin core that check the compliance factor of these transactions.

In the program, we simply call these functions. For example, the code for basic transaction tests exists in a function called `CheckTransaction`. This function is called in our code `CheckTransaction1`.

As always, we have made no major change to the original code, the comments are also not touched. The variable `transaction` stores the actual bytes of the transaction, in this case the transaction hash starts with 23b3.

```
bitcoin-cli getrawtransaction
```

```
23b397edccd3740a74adb603c9756370fafcde9bcc4483eb271ecad09a94dd63
```

The bytes returned by the command `getrawtransaction` is assigned to the variable, `transaction`. All serialization is done in the Bitcoin code using a class called `CDataStream`. The constructor in this class hates the old C type pointer to char and insists on taking the newer vector of chars. The constructor in class `CTransaction` uses the word `deserialize` or `{}` as the first parameter, take your pick. The `ParseHex` function takes a character array and returns a vector of chars. The last two hash defines are insignificant for now.

A `CTransaction` object, `tx` is finally created and it represents a Bitcoin transaction.

This class has 4 useful members; 1) `version`: The transaction version, 2) `vin`: a vector representing multiple inputs of type `CTxIn`, 3) `vout`: a vector representing multiple outputs and finally 4) the rarely used `nLockTime`.

Our transaction starting with 23b is accessed using these 4 fields.

An empty state object, `state` is created and then function `CheckTransaction1` is called. The function is given the transaction object, `tx` and state object, `state`. The last parameter is a hard-coded value of `true`, it checks for duplicates in the input. The return value of the `CheckFunction1` impacts the behavior of the macro `BOOST_CHECK`. The program checks only those transaction errors that do not depend on, as the comments say, any context. These are also very simple error checks, from our perspective.

Every vector has a member called `size` that gives a count on the number of members present in the list/array/vector. The helper function called `empty`, checks if the size of the vector is 0.

The basic rule is that every transaction must have at least one input and one output. Someone brings in money and someone spends the money brought in. If an error occurs, a value of `false` is returned and the fields of the state object are filled up with the type of error that took place.

The function `GetSerializeSize` in our case, simply counts the bytes that make up the transaction, 158. Then we do something odd, only in our opinion. The size of the transaction is checked. Is it greater than a macro `MAX_BLOCK_SIZE`, which is 1,000,000 or a Megabyte, the max block size?? We always thought that the size of all the transactions in a block would be less than the max block size of 1MB. We are assuming that this is the only transaction in the block. All this changes in segregated witnesses.

In the outputs, the amount in Bitcoins must always be positive, 0 is allowed. There is a check for this contingency as well, which insists that the value must not be greater than a macro `MAX_MONEY` or 2100000000000000; the upper limit of Bitcoins.

A transaction can have multiple outputs so a running total of all Bitcoins in a transaction is maintained in a variable called `nValueOut`. This cumulative total of all transactions outputs must not exceed `MAX_MONEY` and the function called `MoneyRange1` performs this check.

The next check is on duplicate Inputs (sort of) and not Outputs. A vector can store any object, including a `COutPoint`. A `COutPoint` consists of a 32-byte transaction hash and an output index. The transaction output in the input is represented in this manner. No one can refer to the same hash value and output index twice in the same transaction's input.

The for loop scans every input. The object `txin` represents a valid input in each iteration. The `COutPoint` object `prevout` is inserted into the vector `vInOutPoints`. In this manner, duplicate transaction inputs are checked. As pointed out earlier, the inputs cannot have the same transaction hash and output index, as it uniquely identifies an output that brings in bitcoins. We can never spend part of an output. This way we check for a double spend in a transaction. This check, as the comment says, is slow and hence is normally not performed.

Finally, there are a different set of rules for a Coinbase transaction or the first transaction in a block. For a non-Coinbase transaction, the `prevout` object must not be null, it cannot point to a non-null transaction hash and an output index. For a Coinbase transaction, the dummy script signature or `scriptSig` field must have a length which is less than 2 or a length larger than 100, which befuddles us. The effective check is that the `scriptSig` must be greater than 3 bytes.

These are some very basic checks, but a good start. We devote the second last chapter of this book to more tests, we cover these checks once again plus some more.

The Bitcoin Core developers created some zillion checks and conditions which we can incorporate in our code. They basically created a json file in the `test/data` folder and placed all the data for error checks in these files. In the program, we manually initialized the transaction variables. The Bitcoin tests read these json files line by line as an array. The transaction data is stored in these json files and a corresponding header file with the same name is created with the data stored as an array in the same data folder.

We were unaware of the fact that there are many different types of hashes computed on a transaction. This can be learnt only after reading the tests. These tests give an insight to the Bitcoin source code. It gives a vague idea as in where to start in our attempt to understand the source code.

The tests written in C++ are at a very higher level, the Python tests are at a very lower level. We can write a 1000 pages just on Python and C++ tests but debugging code is not our forte. We abhor macros in code. So, we have cleaned up the macros in the above code.

Add the lines given below in the file `Makefile.test.include`

```
test/mukhi_tests.cpp \
test/vijay_tests.cpp \
test/util_tests.cpp
```

It adds the file `vijay_tests.cpp` to the list of files to be compiled.

Removing all the Macros Used in the Test Suite

```
vijay_tests.cpp
#include "test_bitcoin.h"
#include <boost/test/unit_test.hpp>
bool glibc_sanity_test();
static boost::unit_test::ut_detail::auto_test_unit_registrar vijay_registrar( "vijay1",
"1vijay_tests.cpp", 23, boost::unit_test::decorator::collector::instance() );
static void mukhi_invoker()
{
    do
    {
        printf("While Only Once\n");
        ::boost::test_tools::tt_detail::report_assertion ( (glibc_sanity_test() == true),
        (::boost::unit_test::lazy_ostream::instance() << "Vijay Mukhi is a embecile15"),
        ::boost::unit_test::const_string( "3vijay_tests.cpp", 10 ),
        static_cast<std::size_t>(700), ::boost::test_tools::tt_detail::CHECK,
        ::boost::test_tools::tt_detail::CHECK_MSG , 0 );
    }
    while( ::boost::test_tools::tt_detail::dummy_cond() );
    bool a = glibc_sanity_test() == true;
    printf("Hell1 %d\n" , a);
}
static boost::unit_test::ut_detail::auto_test_unit_registrar mukhi_registrar(
boost::unit_test::make_test_case( &mukhi_invoker, "mukhi1", "2vijay_tests.cpp",
98 ), boost::unit_test::decorator::collector::instance() );
static boost::unit_test::ut_detail::auto_test_unit_registrar end_suite11_registrar11( 1 );
```

Output

```
Running 1 test case...
Entering test module "Sonal Test"
1vijay_tests.cpp:23: Entering test suite "vijay1"
2vijay_tests.cpp:98: Entering test case "mukhi1"
While Only Once
In glibc_sanity.cpp
3vijay_tes:700: info: check 'Vijay Mukhi is a embecile15' has passed
In glibc_sanity.cpp
Hell1 1
2vijay_tests.cpp:98: Leaving test case "mukhi1"; testing time: 87us
1vijay_tests.cpp:23: Leaving test suite "vijay1"; testing time: 95us
Leaving test module "Sonal Test"; testing time: 159us
```

Macros hide the complexity of the boost code and hence it becomes difficult to understand and explain the above code. Please ignore this program if it looks difficult. Life would be better if nothing was hidden.

The single biggest change in Bitcoin Version 0.15 is the addition of a zillion new tests. The best way to have your name amongst the contributors of Bitcoin source is to write tests.

CHAPTER 32

Ethereum Solidity

This chapter and the next two chapters compare two different approaches to blockchain technology. It is still unclear and difficult to define a blockchain.

Mr. Satoshi Nakamoto wanted to create a crypto currency where no entity had any kind of control. With this clear vision, there were a series of technologies created and that is collectively called blockchain today. It surely was not a call taken one fine morning to invent blockchain technology.

On the other hand, Vitalik Buterin, the co-founder of Ethereum woke up in the middle of a night (this is not fake news) and said he wanted to build a World Computer. So, he created a different set of technologies for the problems he wanted to solve with his World Computer. These technologies are also termed as a blockchain.

These theories remind us of stories like 26 blind men and an elephant/blockchain. Everybody has a different view when defining it. We are not here to take any sides. Our approach is to understand these blockchain technologies and find out how different Ethereum is from Bitcoin. Most people term Ethereum as Bitcoin 2.0.

We assume you have successfully installed the geth program from the Ethereum website. Ethereum or its client program, geth, runs on Windows, Mac OS and all variants of Linux. Operating Systems should not be of any concern here. For the record again, we use a Mac and sometimes Ubuntu Linux under a virtual machine on our Mac. For a Mac and brew, it is a two command install.

Ethereum uses different clients unlike Bitcoin, which has only one standard client. Bitcoin uses the singular, Ethereum uses the plural. The latest Ethereum client is written in the rust programming language. The Ethereum folks prefer using geth, which is written in the Go programming language.

The Bitcoin chapters focused on every file and various components created by Bitcoin on our disk. The focus of this chapter is more on the Solidity programming language. This is a higher-level chapter; the next two chapters focus on the lower levels. It is our suggestion that you read all three chapters, only to understand Bitcoins better.

The core idea of Ethereum over Bitcoins is to run programs on computers that we do not own or even know or care about. Ethereum uses concepts like running a World Computer.

The folks at Ethereum pay their debt to Bitcoin so it is pointless comparing who is better. Ethereum recommends that all our code be written in a programming language called Solidity which looks and feels and walks like JavaScript. JavaScript will be the third language used in this book, though not the last.

If you want to run a program on computers all over the world, you must pay for it. The currency used is called Ether or ETH. The problem is that while learning ETH, we will be running millions of programs all the time which can create a big hole in our pockets.

Luckily for us, the folks at Ethereum created a network called testnet which is similar in all aspects with the main network but uses Ether with no value at all. Borrowed lock stock and barrel from Bitcoin !!! After some time, we will shift to the real Ethereum network with real Ether. For this chapter, it does not matter.

The first task is to download all the testnet transactions that have taken place, or the blocks having transactions.

```
geth
geth testnet
```

The main live Ethereum network is called Frontier and the corresponding testnet, at the time of writing this book, is Morden. Morden ran from July 2015 to November 2016. Then Morden hit a roadblock and it was rebooted though not declared dead on arrival. Modern science came in and Ropsten was born as the new testnet.

Then, there was another breakdown in Feb 2017. A massive attack was launched on Ropsten. With names like Ropsten, you make yourself a sitting target. Parity written in rust is a competitor to geth and they came out with the Kovan network as an emergency measure. The geth client is not allowed to work with Kovan. The kingdom hits back and comes with another one, Rinkeby one more testnet.

This program should take about an hour or a day depending upon the speed of your internet connection. The output says blocks downloaded are around 30000, the number of blocks when writing this chapter. The number will be much higher when you read this chapter.

Our work starts now. Start geth again as in

```
caffeinate -i geth console 2>>/dev/null
caffeinate -i geth --testnet console 2>>/dev/null
caffeinate -l get -rinkeby
```

The caffeinate program used in the earlier chapter is brought in again. Our client is the geth program. The console command talks to Ethereum core using geth. The only problem with geth is that it is quite verbose and displays too many things which come in our way.

The 2>> represents the standard error output. The screen is represented by the number 1 and the keyboard by file handle 0. Geth writes to the output specified.

When we write to the file /dev/null, the data gets sent to nowhere. Therefore, we see a single greater than sign > waiting for us to express our feelings.

```
> eth.accounts
[]
```

There is a free object/word/command called eth that has a variable or property called accounts. JavaScript like tools has many freebies. An account in geth is a like all bank accounts, it stores the ether or money we own. We have Bitcoin addresses and we have Ethereum addresses. As we are running geth the first time, the resultant list has no values. [] represents a list or multiple values separated by a comma,.

```
exit
```

The command exit quits out of geth.

```
geth --testnet account new
```

Output

Your new account is locked with a password. Please give a password. Do not forget this password.

Passphrase:

Repeat passphrase:

Address: {0xb774cf0d91bae0dc2fe8ba7e3138dd5afe356ba2}

As a rule, if the testnet option is not specified, a connection is made to the existing Ethereum network.

A password is a must for accounts created in geth. The password is given twice for confirmation. It then reveals our Ethereum address or bank account number or Bitcoin address. Your mileage will vary obviously. Ethereum is as different from Bitcoin like chalk is from cheese.

Now we go back to Ethereum again.

```
caffeinate -i geth --testnet console 2>> /dev/null
eth.accounts
["0xe844b998e8f881c5779ba63e37fc4daec5c6f2d7"]
```

Exit and quit out and then run the following command.

```
geth --testnet account new
Your new account is locked with a password. Please give a password. Do not
forget this password.
Passphrase:
Repeat passphrase:
Address: {0xffd0ea6656c61423210adbd9e4801d5854334649}
```

This Ethereum address is 40 characters large.

```
Back to geth
caffeinate -i geth --testnet console 2>> /dev/null
> eth.accounts
["0xbf254211c6b0a6a2882e269008b0024f3e6b5270",
"0x08f3556f206e352303857e1567c307b06a3a9e22"]
```

We have created two accounts or testnet Ethereum address by running account new twice. The actual addresses will be different when you run the account new option. That's why chapters like these become difficult for you to understand.

It is very easy to extract each member or element of the list using [] brackets and the number or index of a specific item in the list. Standard study for all lists.

```
eth.accounts[0]
"0xbf254211c6b0a6a2882e269008b0024f3e6b5270"
> eth.accounts[1]
"0x08f3556f206e352303857e1567c307b06a3a9e22"
```

Now to look at the amount of money in our Ethereum addresses.

```
eth.getBalance(eth.accounts[0])
0
eth.getBalance(eth.accounts[1])
0
```

This object eth has many functions like getBalance which disclose the number of ether in our Ethereum addresses. This is like the testnet on Bitcoin, this is not real ether. There are however websites that convert Bitcoins into ether.

Since the account is newly created, there are no ethers. Though we would have liked it if the folks at Ethereum gave us some free ether.

There are many ways to get free ether, one way is to become a miner and create ether out of thin air. The wait is too long here. Then, there are several sites that give you free testnet ether. The first site is:

`https://zerogox.com/ethereum/wei_faucet`

Write you Ethereum testnet address and thou will get free ether.

The second site is

`http://faucet.ma.cx:3000/`

A third and better way is to run the following program.

`sudo npm install -g ethrain`

If you do not have npm installed, use homebrew. Then run the following command multiple times.

`ethrain <your Ethereum address>`
or
`ethrain 0xb774cf0d91bae0dc2fe8ba7e3138dd5afe356ba2`

The above command gave us ethers in the year 2016 but it failed in the year January 2017 but once again worked in October 2017. Anyways, we then used the following two websites to collect free ethers. Anything free is not sustainable.

`https://test.ether.camp`

In the chrome browser only, enter the following:

`http://faucet.ropsten.be:3001/`

We hope these websites are still running when you read this book. Remember, a testnet is not meant to live forever. Use Google to find working testnet faucets. After all, it is social service, nobody is making any money to give you free testnet ether.

```
eth.getBalance(eth.accounts[0])
10000000000000000000
```

Now that we have collected plenty of ethers to spend, let's get down to writing code in the Solidity programming language. Anything starting with a > sign means enter the command in the geth console. The testnet output will be displayed only when it is different from the real Ethereum network.

```
>s
ReferenceError: 's' is not defined
    at <anonymous>:1:1
```

Geth has no idea what s represents, hence the error message. Sadly, error messages are never written in English :(.

```
> s = 10
10
```

Geth echoes the current value of s. Set the value of s to a number, 10. Now, Geth displays the value of the variable or object s, which is 10.

```
> s
10
```


Now `s` displays an output, 10. The variable `s` is not an error anymore.

```
> s = "hi"
"hi"
```

The object `s` is set to a value of "hi" which is a string. There are no complaints here and we see hi displayed. Strings can use either single or double inverted commas, but geth always encloses them in double quotes.

```
> s
"hi"
```

Now the value of object `s` changes to hi. This is the concept of a type-less variable. It reminds us of Python.

```
> s = [10, 20]
[10, 20]
> s
[10, 20]
```

The property accounts in `eth` object is a list. To create a list type, simply place values in `[]` separated by commas.

```
> s = {mukhi:10, vijay:"hi"}
{
  mukhi: 10,
  vijay: "hi"
}
> s
{
  mukhi: 10,
  vijay: "hi"
}
```

It is possible to create complex types called dictionaries or structures in Solidity. Simply, start with a `{}` and then place key-value pairs, separated by a comma. The key ends with a colon and it is followed by the value. In the above snippet, there are two keys, `mukhi` and `vijay` with the values 10 and hi.

The object `s` by itself displays all the key-value pairs. Or, you can be selective and display the key you want.

```
> s.mukhi
10
```

To access individual members, simply use the object name, a dot and the name of the key. Most programming languages including Python and not C++ follow the same syntax.

```
> s = {mukhi: {vijay:10}}
{
  mukhi: {
    vijay: 10
  }
}
> s.mukhi
{
  vijay: 10
}
```

```
    vijay: 10
  }
  > s.mukhi.vijay
  10
```

This dictionary can hold many key-value pairs, a value can also be another dictionary. For example, you can have a key called mukhi in a dictionary whose value is another dictionary type. To access the innermost key, start with the dictionary name, s, and then the first key name, mukhi, followed by the inner key name vijay. All names separated by a dot.

```
> var c = eth.contract([]);
undefined
```

In the console, the same free object eth is used to call a function called contract and it is assigned an empty list. The value returned is a contract object called c. The var keyword is optional. Every Solidity program uses the var keyword when creating an object, so we do the same. You can see the key-value pairs this variable c brings to the table.

A lot of us old timers who were raised on the C programming language place a semicolon; at the end of the line. The same tradition is followed here too.

The list given to the contract function is also optional. In Geth, enter c and eth. The output is a long list of properties and functions. The list for eth is quite large, we have displayed only two lines from the output. They both, kind of give the same output.

```
>c
{
  abi: [],
  eth: {
    accounts: ["0x2c80814d41030380606fd8e2fb988dedd6aae1d9",
      "0x0be79512b356cffe8b73e3c8d9555f5c951b1c0b",
      "0x8c4e0c160a0d8c17dfba4b04e061b42767e4abe9"],
    blockNumber: 2690459,
    coinbase: "0x2c80814d41030380606fd8e2fb988dedd6aae1d9",
  }
  > eth
  {
    accounts: ["0xb774cf0d91bae0dc2fe8ba7e3138dd5afe356ba2"],
    blockNumber: 363282,
    coinbase: "0xb774cf0d91bae0dc2fe8ba7e3138dd5afe356ba2",
    compile: {
      lll: function(),
      serpent: function(),
      Solidity: function()
    },
  },
}
```

The output displayed above shows that the word accounts is a key in the eth dictionary object. The key blockNumber denotes the current block number, the one that is active at the time of writing this chapter. The compile keyword notifies that our copy of geth can support three programming languages Solidity, lll and a python clone called serpent.

So far, a contract object c is created with a zillion key-value pairs.

```
var v = c.new({from: eth.accounts[1] , data:""})
var v = c.new({from: eth.accounts[0] , data:""})
account is locked
```

var v is another object like c and it is created when the new function from the contract object c is called. The new function is given one dictionary object only.

This dictionary has two key value pairs, one is a field called From which has its value as the Ethereum address and next is a key called data which is an empty string, for the time being.

We thought that we had written our first smart contract but were mistaken. We get an error saying that the account is locked. In Byzantium, the error message is more civil, it changes to authentication needed: password or unlock.

```
personal.unlockAccount(eth.accounts[0] , "zzzzzzzz" , 60 * 60 * 24)
true
```

To unlock it, one more free Solidity object, personal, comes in the picture. This object has a function unlockAccount, which takes three parameters. The first is the Ethereum address of the account to be unlocked. In our case, it is the first Ethereum address in our list for the testnet or the real network.

The second parameter is the password used when creating the account, ours is not zzzzzz. The third parameter is the time in seconds, duration as to how long the account stays unlocked. The calculation is in seconds and represents a full day. Please bear in mind, that when you logout and login again into geth, the account gets locked again. Ethereum locks our accounts without our permission.

```
var v = c.new({from: eth.accounts[0] , data:""})
undefined
```

Now we get no error but simply an undefined displayed on the screen, some progress. So, let's display the value of variable v.

```
> v
{
  abi: [],
  address: undefined,
  transactionHash: "0x9d4a05679965016a40fac90472e40807dddef6434495d358d180ffd26ef31c37"
}
v
{
  abi: [],
  address: undefined,
  transactionHash: "0xa5a852673b8356830ff732d5d140446d68588da146863a90e5db259dccf4e34d"
}
```

v is a dictionary with three keys, abi, address and transaction. We simply created a new Ethereum transaction or better still a smart contract.

It's time to verify our claims. Ethereum like Bitcoins, has many web explorers that showcases all the Ethereum real and testnet transactions. Open any browser and type the following URL.

```
https://etherscan.io
or
https://ropsten.etherscan.io
```

Then in the text box copy, paste the above transaction hash starting from 9d4a in the first URL. This is the real ethereum network browser. The second URL should be pasted in the Ropsten testnet browser. The 0x at the beginning has no relevance, whatsoever. The first transaction hash has a date time stamp of Feb 2017 and the second, a date time stamp of Oct 2017. We expect you to follow the second output as the first one will burn a big hole in your pocket. We have both, take your pick.

Normally it takes about 20 seconds for the transaction details to show up, at times more. A little patience helps, as at times there can be a wait of a minute or three also.

In the web browser, we see the From: column with our Ethereum address starting from 0xbe75 or 0xbf25 (since we created this transaction). These are our ethereum addresses, you will never see such addresses.

The Block height column displays the block having this transaction. The block height in our case is 3224383 or 1883141. The Value column is 0 only because this transaction is a smart contract, it is not sending money or ether to another Ethereum address. The input data column at the bottom of the webpage is blank as our key named data is empty or has a null string.

The new function actually created a smart contract on the Ethereum network with no code to run.

Please note down the nonce field, as it will keep increasing by 1 each time your Ethereum address runs a smart contract. Presently, its value is 0.

The Gas columns refers to the amount paid in ether to the miners who run the Ethereum network. The more complex the smart contracts, the more gas or money you need to pay.

In our case, we used up 53000 as gas. The gas price to pay is shown as well. Nothing in life is free even for an empty contract that does nothing.

Patience will take you every far. Always wait for a minute or two before screaming murder. Transactions take a minute or two to show up. Not like Bitcoins 10 minutes plus and counting. Many a times in its existence, the testnet is reset and all our balances and transactions go into the ether or into a black hole. So, you start from scratch then.

```
> var v = c.new({from: eth.accounts[0] , data:""})
undefined
> v
{
  abi: [],
  address: undefined,
  transactionHash: "
0xd504d2d4be2ef7a6f1a592d6a373311e9da0a7b44d6a47033e6b9183a7934d63"
}
```

The same code is executed as we have lots of ether to spend. The focus is on the nonce field, it has a value of 1. Further, it confirms that we have sent two transactions with this Ethereum address. In cases where we see only one output, we do not show you the output in testnet, but you must run every line of program you see.

As all the rules laid out by the Ethereum folks are being strictly observed, the address key is left undefined. This is the price you pay for breaking rules.

To summarize our achievement, we created an empty contract object, c with no data. Then an object v is created by only specifying our Ethereum address. Within 20 seconds, we have a smart contract or transaction using our name on every computer or miner, willing and able to execute our code on the Ethereum network. YAY. Thus, it is a fallacy to ever compare the Bitcoin and the Ethereum blockchain, as they are as different as chalk and cheese.

This smart contract of ours has been running for 237 days and not complaining. This could be an Initial Coin Offering or an ICO where you create your own digital token or money. At the end of this chapter, you will get a feel for how a ICO can be created using a smart contract.

Let's now take things one step at a time. We can afford Ethereum testnet ether as there is no money paid here. Imagine spending real money on a smart contract that does nothing.

```
> var v = c.new({from: eth.accounts[1] , data:"0xff"});
undefined
```

At times, copy cut paste will not work.

```
> v
{
  abi: [],
  address: undefined,
  transactionHash:
    "0xbc64e84eb09ff5052cd0a69a7499d9ccd3ace1bd9b09cb98d705bd92d973a0e7"
}
```

This is for testnet, in future the second output is for testnet.

```
transactionHash:
  "0x8f9f5413cb8ed40723b3d2eb358bd1a88f1aa87b2a0dc470f59ad700570e328c"
```

Lots of Patience required here. In the Bitcoin world, the mining time of each block is officially 10 minutes, in Ethereum it is 15 seconds. At times, the wait can be for 3 minutes before the transactions shows up in the explorer. The etherscan explorer is also slow. The changes in nonce does not affect the nonce value of the earlier transactions.

There are three facts highlighted with this transaction hash. The Input data shows a value of 0xff, the value of the data field. The nonce is now 2. And, there is a Contract warning in red color. Ditto for the testnet. A string 0xff is added to the data key.

This proves that any value given to the key called data will show up on all computers in the Ethereum network, provided it holds a transaction or a smart contract. So far, both networks are functioning in the same way. Later, we will explain that code of the smart contract can be assigned to the data key and it will get executed on these zillions of computers. This code will soon be a trillion lines long.

So far, we have successfully created some kind of a smart contract. The only missing piece of information is assigning actual bytecodes, which will then be executed.

Let's first check if there any Ethereum compilers installed on disk.

```
> eth.getCompilers()
[]
```

The function getCompilers returns all the compilers for various Ethereum languages installed on our computers. We have zero. So, install Solidity by using the command

```
$brew install solidity.
> eth.getCompilers()
["Solidity"]
```

When we ran the same code earlier.

```
cc = eth.compile.solidity("contract mukhi {function abc(){}")
{
  mukhi: {
    code:
    "0x6060604052346000575b605d8060166000396000f300606060405263ffffffff60e0600
    20a6000350416639227793381146022575b6000565b34600057602c602e565b005b5b56
    00a165627a7a723058209517cfe8b354c9ccbfc29c2534e1bdbe19a4443375ccbfdb37
    bce611d37bb70029",
    info: {
      abiDefinition: [{...}],
      compilerOptions: "--bin --abi --userdoc --devdoc --add-std --optimize -o
      /var/folders/zw/g899mbvx7g58zc_zl2x69ljw0000gn/T/solc488030563",
      compilerVersion: "0.4.8",
      developerDoc: {
        methods: {}
      },
      language: "Solidity",
      languageVersion: "0.4.8",
      source: "contract mukhi {function abc(){}",
      userDoc: {
        methods: {}
      }
    }
  }
}
```

When we ran in the beginning of the year 2017.

```
> cc = eth.compile.solidity("contract mukhi {function abc(){}")
{
  <stdin>:mukhi: {
    code:
    "0x60606040523415600b57fe5b5b605e8060196000396000f300606060405263ffffffff60
    e060020a6000350416639227793381146020575bfe5b3415602757fe5b602d602f565b00
    5b5b5600a165627a7a7230582074a3c649e72264609d1ba9967852b72377f1d6cffaf6ef6
    1753039d05b8bae680029",
    info: {
      abiDefinition: [{...}],
      compilerOptions: "--combined-json bin,abi,userdoc,devdoc --add-std --optimize",
      compilerVersion: "0.4.9",
      developerDoc: {
        methods: {}
      },
      language: "Solidity",
      languageVersion: "0.4.9",
      source: "contract mukhi {function abc(){}",
    }
  }
}
```

```

        userDoc: {
            methods: {}
        }
    }
}

```

We have been tearing our hair apart and screaming and shouting endlessly. That's because all the above code, we tried and tested earlier this year, breaks down today, it does not work. Code stops working.

The folks at Ethereum make a change in their binaries and forget to update their own tutorial called Hello World or greeter.

The object called cc, now, does not have a field called mukhi but `<stdin>:mukhi`. Not visible anywhere in the testnet output !!!

To display the code member, we must use the new format

```

cc[“<stdin>:mukhi”].code
0x60606040523415600b57fe5b5b605e8060196000396000f300606060405263ffffffff60
e060020a6000350416639227793381146020575bfe5b3415602757fe5b602d602f565b00
5b5b5600a165627a7a7230582074a3c649e72264609d1ba9967852b72377f1d6cffaf6ef6
1753039d05b8bae680029

and not as
cc.mukhi.code

```

A compiled object called cc is created using the Solidity function present in the compile object. The string given to this function is generally code written in the programming language, Solidity. The keyword contract is part of the Solidity programming language and the brackets are part of syntax. My family name mukhi is the name of the contract here.

Every smart contract starts with the reserved word contract and then the name of the contract, as in contract mukhi. The abc function in the contract object takes no parameters, returns nothing and does nothing.

The compiled object cc has a key called mukhi, the name of the contract. The value assigned to it is a set of key value pairs. The focus here is on the key called code which is simply made up of numbers.

The output for the key code shows only code bytes, but these bytes may change over time with changes in the compiler versions. These changes have nothing to do with the testnet or real Ethereum network.

Some theory. In the last century, a compiler would take a program and convert it to a series of numbers or opcodes or machine language instructions for a certain microprocessor only. Therefore, a program compiled for an Intel chip that powers your servers and desktop would never run on an ARM chip, which powers the mobile phones.

Sun invented the java programming language with a very interesting concept. They created a virtual or a non-existent processor. The java compiler looks at the source code written in Java and creates bytes or bytecodes that would execute only on this imaginary machine. This machine was called the Java Virtual Machine or JVM.

The java compiled code would run only on the JVM and not on any computer in the world. The JVM had code for every OS/chip combination which converted the java bytecodes to the actual hardware it was running on. All this did was delayed the problem to tomorrow.

Ethereum follows the same rules, it converts code written in the Solidity programming language to bytecodes that run only on the Ethereum virtual machine. These bytecodes get converted to the Intel chip running on our Mac, as we are miners.

The key named data is assigned the bytecodes that will be executed on the Ethereum virtual machine. Every computer in the world that is part of the Ethereum network will execute these bytecodes irrespective of the chips and Operating System they use. The problem with compiled bytecode is that all the variable names and contract names get removed from the code.

A list of function names that constitute our contract is called an Application Binary Interface or ABI in short. The task now is to forward these names, namely variable names, contract names, function names, that make up our contract, to the Ethereum contract function.

The code that manages our ICO will be placed in functions like abc.

In Byzantium, we get an error as they have removed the getCompilers function and compile. You can check this out for yourself using Google. One of the many reasons why we run our code multiple times. Now what do we do. So what next, scrap the chapter. No.

If you want to actually run this code in geth, do the following. You create a file a.sol in the current folder which looks like.

```
contract mukhi {  
    function abc() {  
    }  
}
```

Next run the solc compiler as

```
solc --bin a.sol
```

It will display the same code bytes that you can copy, cut and paste. Difficult but this is what happens when you change the rules of the game all the time. The Ethereum developers just removed compiling from geth. The tutorial in October 2017 still uses compile. This is what importance the Ethereum developers give to people who read their tutorials and people like us who write books.

You could also use the remix online IDE at <https://ethereum.github.io/browser-solidity/> and click on Details after writing your code in big IDE window. The column name bytecode is what you are looking for.

But, all is not lost. In chapter 38, we continue from where we stopped and we will compile our solidity code outside of geth without you doing any copy, cut or paste. For the record, all the tutorials on the Internet on solidity and geth have stopped working. As we said before, read the forum posts on how upset people are. The Ethereum developers make changes and then do not care about their implications. We are not protesting the changes made by the developers, but please update your own tutorials at least.

So please read on without trying the code, for the sake of continuity we will see you in chapter 38.

```
> cc.mukhi.info.abiDefinition  
[  
  {  
    constant: false,  
    inputs: [],  
    name: "abc",  
    outputs: [],  
    payable: false,  
    type: "function"  
  }  
]
```

The key abiDefinition is a list, a dictionary with 6 key value pairs. The most important one is the key called name having

a value of abc. The value assigned to this key is a function. The inputs and outputs are lists, but for now they are empty. If there were two functions, then the list would have two dictionaries.

The name of the contract, mukhi is not visible in the abi definition. Left us disappointed.

Today you run solc as

```
solc --abi a.sol
```

Now let's get back to our older code.

```
> var c = eth.contract(cc.mukhi.info.abiDefinition)
undefined
```

This chapter on Solidity is to emphasis on the fact that Ethereum has an entire programming language along with it, unlike Bitcoin. Our goal is met here.

Now let's execute our contract with real bytecodes.

```
v
{
  abi: [{
    constant: false,
    inputs: [],
    name: "abc",
    outputs: [],
    payable: false,
    type: "function"
  }],
  address: undefined,
  transactionHash:
    "0xebedb23a166fb06734666b09517e07c8442efda3ff2b0ebed76d793f7ca758e4"
}
```

At times, an error can occur stating Nonce too low. The reason here is that the Ethereum state is out of sync. So, we leave Ethereum alone for some time and it comes back to normal.

Now there is a transaction hash value, a key called abi containing one function abc and nonce with a value of 4 in the testnet output.

Let's see this transaction in our real/testnet browser explorer. The transaction details are shown, as before. Hang on, it may take time to mine the transaction, the wait time will be in minutes and not seconds. To our surprise, the webpage shows 11 confirmations. In October 2017, there are 1133599 confirmations.

On the testnet webpage, the input data starts with 0x6060 and ends with 5b00. Yay, our code has been registered with all the Ethereum World Computers. Now, everyone in the world can see our code on the main Ethereum network.

But if you yet want to run this code in October 2017, this is what you do.

```
zz =
[{"constant":false,"inputs":[],"name":"abc","outputs":[],"payable":false,
"stateMutability":"nonpayable","type":"function"}]
```

First is the abi definition in a variable called zz. This is after running solc.

```
var c = eth.contract(zz)
```

Then is the contract using the abi definition.

```
bytecode = "0x6060"
```

Then, the byte code in a variable called bytecode but not the original.

```
var v = c.new({from:eth.accounts[0] , data: bytecode})
```

Executing the contract

```
v
{
  abi: [{
    constant: false,
    inputs: [],
    name: "abc",
    outputs: [],
    payable: false,
    stateMutability: "nonpayable",
    type: "function"
  }],
  address: undefined,
  transactionHash:
    "0x22385e9ae602d51feaf929b356d46f11f7895cb54a18ac70acb2ff7bf0b29dbf"
}
```

Confirm the transaction hash. The nonce is 4 so we do not run just once. Too much effort, just check our answers.

```
> v.abc()
```

The above command gives an error. Our earlier function abc returns no value.

So, lets modify the code. After the name of the function, the returns keyword is used and followed by the data type of the values to be returned. We return a string, vijay which is my name with the return keyword.

```
> cc = eth.compile.solidity("contract mukhi {function abc() returns (string) {return  
'Vijay' ;}}")
```

```
var c = eth.contract(cc.mukhi.info.abiDefinition)
```

```
> cc.mukhi.info.abiDefinition
```

```
[[
  constant: false,
  inputs: [],
  name: "abc",
  outputs: [{
    name: "",
    type: "string"
  }],
  payable: false,
```

```

    type: "function"
  }]

```

Since one value of string type is returned, the outputs list has one item only.

We run again

```

var v = c.new({from:eth.accounts[0] , data: cc.mukhi.code})
> v
{
  abi: [{
    constant: false,
    inputs: [],
    name: "abc",
    outputs: [],
    payable: false,
    type: "function"
  }],
  address: undefined,
  transactionHash:
    "0x24529fe73b048cf8ad410c8d526a05085a596c2a9df0fa72c167f96c2218185e"
}

```

On the main network, our transaction hash starting with 24529 is found.

Enter this value in an explorer and a gas error is reported.

Warning! Error encountered during contract execution [Out of gas]

The default gas or the price paid to the miners to run our code was not enough.

Something wrong somewhere. We are now going to write our code in in big file and run it at once. So, make sure that the following file is created in the folder running geth.

Using the Log Function in a JavaScript File

```

ch3201.js
console.log("hi")
i = 10
console.log("Bye " + i)
> loadScript("ch3201.js")
hi
Bye 10
true

```

The object console like eth is freely available and the function log writes or displays text on the geth console. Using the log function from the console object, a value of Hi is displayed first. Then a variable i is created and assigned a value of 10. To display the value of this variable using the log function, we simply use the + sign with the name of the variable, i.

The log function comes in handy when data is required from the contract. The loadScript function executes a series of Solidity commands at one go.

Always end a line with a semicolon even though it is optional.

Running a Contract Using a JavaScript File

```
ch3202.js
var s = 'contract mukhi ' +
'{ function abc() constant returns (string) ' +
  '{ return "Vijay";}' +
  '}'
var cc = eth.compile.solidity(s)
var c = eth.contract(cc.mukhi.info.abiDefinition);
var v = c.new({from:eth.accounts[0], data: cc.mukhi.code , gas:100000 })
> loadScript("ch3202.js")
true
```

In future, you will call the loadScript function to see the output.

Here, the string s is spread over multiple lines for clarity. The contract name once again is not important. The javascript code has a keyword constant before return, which is not ignored by the Solidity compiler, but we will ignore it.

The variable cc is the compiled object. Only the abi and the code keys are significant. In the new function, the keyword gas is added and it is given a value of 10000, the minimum value to execute our code.

```
> v
{
  abi: [{
    constant: true,
    inputs: [],
    name: "abc",
    outputs: [{...}],
    payable: false,
    type: "function"
  }],
  address: undefined,
  transactionHash:
    "0xc7786df0795cbd32e34f96fb931395b1488d812c556f2949dec4afe97937c0ba"
}
```

After a very short wait

```
> v
{
  abi: [{
    constant: true,
    inputs: [],
    name: "abc",
    outputs: [{...}],
```

```

        payable: false,
        type: "function"
    }],
    address: "0xc6cf8e235eb04afcaacd5da3e3ab5474cb2f87bb",
    transactionHash:
    "0xc7786df0795cbd32e34f96fb931395b1488d812c556f2949dec4afe97937c0ba",
    abc: function(),
    allEvents: function()
}

```

The address displayed here and the contract address seen in the blockchain explorer are the same.

```

Testnet
> v
{
  abi: [{
    constant: true,
    inputs: [],
    name: "abc",
    outputs: [{...}],
    payable: false,
    type: "function"
  }],
  address: undefined,
  transactionHash: "0xdf334e769ac535dccc9960dd799c3a29254100c6dce733cec1b21c5e4c1ec94c"
}

```

After 5 minutes, an output is received, which is displayed below.

```

> v
{
  abi: [{
    constant: true,
    inputs: [],
    name: "abc",
    outputs: [{...}],
    payable: false,
    type: "function"
  }],
  address: "0x1b9846779325ed12821d3b0677a46aa915c3094b",
  transactionHash:
  "0xdf334e769ac535dccc9960dd799c3a29254100c6dce733cec1b21c5e4c1ec94c",
  abc: function(),
  allEvents: function()
}

```

Always note down the value of gas used by the miner. A higher value gets you in a block faster. We played with smaller values, therefore got no confirmations.

The address key finally has a value, earlier it was undefined. The value is the one shown with the To field Contract on the webpage. It is the address of our contract in the Ethereum world. This contract address like our Ethereum address, is unique to every contract.

Now our code executes as advertised.

```
> v.abc()
"Vijay"

v.address
"0x1b9846779325ed12821d3b0677a46aa915c3094b"
```

The address of our contract starts with 0x1b98, it can be verified using the testnet explorer this time. Let's also give testnet a fair stab at the output.

```
> eth.getCode(v.address)
"0x606060405260e060020a6000350463922779338114601c575b6002565b34600257600
060605260c0604052600560809081527f56696a617900000000000000000000000000
0000000000000000000000000000000060a052602060c0908152600560e0819052819061010090
60a09080838184600060046012f1505081517afffffffffffffffffffffffffffffffffffffff
ffffff1916909152505060405161012081900392509050f3"
```

The getCode function returns the same executable code, which was sent across to the Ethereum miners. Once again, it can be confirmed in geth or in the testnet explorer.

Instead of returning values like Vijay we could have a function that returns how many digital tokens have been sold so far. Or the current price of a single digital token and so on.

Let's see how we can run our smart contract. First, create two variables a1 and a2 and share these values with the whole world.

```
> a1 = cc.mukhi.info.abiDefinition
[
  {
    constant: true,
    inputs: [],
    name: "abc",
    outputs: [
      {
        name: "",
        type: "string"
      }
    ],
    payable: false,
    type: "function"
  }
]
```

This is once again a simple list that showcases the functions present in our contract.

```
a2 = v.address
"0x1b9846779325ed12821d3b0677a46aa915c3094b"

> a = eth.contract(a1).at(a2)
{
  abi: [

```

```

        constant: true,
        inputs: [],
        name: "abc",
        outputs: [{...}],
        payable: false,
        type: "function"
    }],
    address: "0x64429ff639bb801d5b01f402acdf92c016a68959",
    transactionHash: null,
    abc: function(),
    allEvents: function()
}
> a.abc()
"Vijay"
>

```

To get access to a contract, the abi and the address of the executing contract is required. The object `a` can now execute the `abc` function. It is only after running the above code on someone else's `geth`, you realize that the abi definition is simply a string giving information on the functions present in the smart contract. Every smart contract is identified by an address. Anyone having access to this address can execute the smart contract.

Therefore, coding for the Ethereum Virtual Machine is very different compared to all we have learnt so far. It deals with code that anyone in the world can execute on all computers in the world. This requires us to think differently. Assuming the code is running on our computers is just an illusion now. We ran this program output on a testnet that is long dead.

How to Kill a Contract

```

ch3203.js
var s = 'contract mukhi { ' +
    'function abc() constant returns (string) ' +
    '{ return "Vijay"; }' +
    'function pqr() ' +
    '{ selfdestruct(msg.sender) ; }' +
    '}'
var cc = eth.compile.solidity(s)
var c = eth.contract(cc.mukhi.info.abiDefinition);
var v = c.new({from:eth.accounts[0], data: cc.mukhi.code , gas:900000 })
>loadScript("ch3203.js")

```

There is one more function, `pqr` introduced in the code. It calls a built-in function `self-destruct`. This function requires an address. The object `msg` has the address of the entity which calls this function, this address is ours.

The key named `abi` has a list of two elements as there are two functions. Next, it is checked if the `abc` function can be called. The `getCode` function displays the bytecodes.

The Main Network

```

> v
{

```

```
abi: [{
  constant: false,
  inputs: [],
  name: "pqr",
  outputs: [],
  payable: false,
  type: "function"
}, {
  constant: true,
  inputs: [],
  name: "abc",
  outputs: [{...}],
  payable: false,
  type: "function"
}],
address: "0x44e572f24ec898bacc7544f5a890805a9e7e53a3",
transactionHash: "0xdd9c4859727c124d6afb95375991b8ede5b22577c2a9dff60748163449cd0ace",
abc: function(),
allEvents: function(),
pqr: function()
}
```

Testnet

```
> v
{
  abi: [{
    constant: false,
    inputs: [],
    name: "pqr",
    outputs: [],
    payable: false,
    type: "function"
  }, {
    constant: true,
    inputs: [],
    name: "abc",
    outputs: [{...}],
    payable: false,
    type: "function"
  }],
  address: "0x0d9e90d070edfbb7405d796278d9e84a34999755",
  transactionHash:
    "0xdf318c2cf377d811553967b182d4f4e1b0cacbe926eb6268b5cf5f0c1d24385d",
  abc: function(),
  allEvents: function(),
  pqr: function()
}
```



```

> v.abc()
"Vijay"
> eth.getCode(v.address)
"0x606060405260e060020a60003504633e7d77d1811460265780639227793314604557
5b6002565b3460025760003373fffffffffff
(part of the output)

```

Here, instead of calling the function `pqr` directly, we do something strange. Every function created in Solidity has a function called `sendTransaction`. This function only requires an account address.

```

Main Network
v.pqr.sendTransaction({from:eth.accounts[1]})
"0x5b3dce4f0498583a630b5417cda32fabddcca0824ab771693d735d7f6c1ec909"

v.pqr.sendTransaction({from:eth.accounts[0]})
"0x3f9df2d7446caa716991d248179a644ef7d4d51c5849ee6a5f5f2c5c05ea5ffc"

```

Function `pqr` is called, the `msg.sender` has our address so the self-destruct function actually kills our contract.

The output is an actual transaction hash value, you can check the same in the explorer. The transaction hash `0x5b` was created by calling the function `sendTransaction`. After killing the contract, wait for some time for the transaction to be mined. Now, run the same `getCode` function. It clearly states that there is no code associated with our smart contract, well actually the code bytes are `0x3e7d77d1` in both nets.

It further goes to prove that the bytecodes that make up the smart contract have been disabled.

```

> eth.getCode(v.address)
"0x"

> v.abc()
new BigNumber() not a base 16 number:

```

The `abc` function also throws an exception. An error as such is a clear indication that the contract has been killed or destroyed.

Again, we are dealing with a free Ethereum testnet. At times, our smart contract gets mined in 20 seconds, at times 20 minutes. The only word here is patience.

There is a dev mode which creates a private network on a computer. It works faster than the speed of light. The displayed transactions hashes or smart contracts in this book will work on your machines then. The hiccup here is that some of this code will not work on your machine either.

One genuine problem here is that no one can infer when our smart contract will be confirmed/mined by the Ethereum network. It is ridiculous to click on the refresh button in the explorer every minute to see some new results. There is a better way.

A Better Way to Create Smart Contracts, Getting Feedback

```

ch3204.js
var s = 'contract mukhi {' +
    'function abc() constant returns(string) {' +
    '{ return "Vijay" ;}' +
    '}'

```

```
var cc = eth.compile.Solidity(s)
var c = eth.contract(cc.mukhi.info.abiDefinition);
var v = c.new({from:eth.accounts[0], data: cc.mukhi.code , gas:900000} , function (e , c) {
    console.log("e " + e + " c " + c.transactionHash + ":" + c.address);
});
```

The only change is the function new. An extra parameter is added to the function. One of the features of the JavaScript language is that an entire code snippet can be entered in place of a parameter.

These functions are called anonymous functions because they do not have a name. Our function takes two parameters, e and c, the second parameter is of type contract. The log function of the built-in console object displays the values in parameter e and the keys, transactionHash and address from the c object.

```
>loadScript("ch3204.js")

Main Network
e null c 0x536e0777e78d9c2569a15688b8da7a9dc4e567e0a431f3cbf4ef43e9d2cb96d9:undefined
true

After the wait, the deluge

e null c
0x536e0777e78d9c2569a15688b8da7a9dc4e567e0a431f3cbf4ef43e9d2cb96d9:0x203
ee8383866657bf663debc505c84d372b49cad

Testnet
e null c 0xfb307e45c7be1670b4c4636694bcc18f1637deb0e1f6fac0320845f82ef12ef0:undefined
true
> e null c 0xfb307e45c7be1670b4c4636694bcc18f1637deb0e1f6fac0320845f82ef12ef0:0x92a80
9e419163a702fd56196d3739b76382d014b
```

When the script is executed, the code within the anonymous function is called instantly. The object c has two fields. The key name transactionHash has the transaction hash and the address key is undefined. This confirms that our transaction is created and sent to the miners to be mined.

After some time when our transaction is mined, the code in our function is called again. Now the address key has a value. It's time to run our code.

Geth calls this anonymous function whenever a certain activity or better still an event takes place. There are two events that take place, one when our transaction is created and second is when the smart contract is mined.

Henceforth, it is advisable to have these anonymous functions monitor and track all events. It makes life so much simpler. So, sit back and let the function inform you about the new events taking place in the system.

Catching Errors in Our Script

```
ch3205.js
var s = 'contract mukhi { ' +
    '}'
var cc = eth.compile.solidity(s)
var c = eth.contract(cc.mukhi.info.abiDefinition);
var v = c.new({from:eth.accounts[0], data: cc.mukhi.code , gas:900000} , function (e , c) {
    if (e)
```

```

    console.log("We have an error " + e);
    else
    console.log("e " + e + " c " + c.transactionHash + ":" + c.address);
  });
  > loadScript("ch3205.js")
  We have an error Error: account is locked
  true

```

As learnt earlier, once you quit out of geth the account gets locked. So, it must be unlocked again. Our code will not run.

So, we must make one small change to the earlier code, we bring in an if statement. Every programming language has an if statement which takes a condition in (). If the condition is met, it is true and it executes all the lines of code between the if and the else statement. If the condition is false, then the code after the else statement is executed. If the number of statements are more than 1, they are placed in {} brackets. This is how we handle errors.

To conclude, Solidity is a full-fledged programming language. All code written in Solidity runs not on one computer but on billions of computers all over the world, not known to anyone. The core idea of Ethereum is to run code written in languages like Solidity. To be precise, the Ethereum computers run bytecodes written for the Ethereum virtual machine, they are not concerned with Solidity or serpent. The crypto currency ether is used to pay for the gas used in driving the Ethereum computers or miners.

Bitcoin is the world most famous crypto currency, please do not confuse it with ether which runs programs on a global scale. The Ether is for paying for your transactions and not for trading. Today lots of people trade in it.

We end this chapter on a sour note and a prayer that the code does not stop working suddenly. At times, it is wiser to use an older version while doing research and learning new concepts. The code works but some functions were removed in later versions.

You could send a man to moon using solidity as it is a programming language that can do anything and everything. Managing an ICO is one of the easiest things.

In one way, this chapter is a total waste. We know that we will make amends in Chapter 38, but none of this is our fault. The fault, dear Brutus lies in our stars. We should be blaming the Ethereum developers but here also there are many people in this long line.

Necessity is the mother of invention. When we first started learning Solidity, water was more expensive than ether. Now nobody can afford both water and ether. Using the real Ethereum network is a waste of money. The biggest problem with Testnet is that when it does not work, all that you can do is simply twiddle your thumbs and wait. This is when seconds become hours.

We have taken a different approach. We create a dev network or a private Ethereum network that no one can access but us. This Ethereum network at times is faster than the speed of light, zero waiting time. Though creating a private Ethereum network had us standing on our head. Once we have learnt a new concept, we try the same on Testnet. Testnet is not used as the network to experiment or to learn but to confirm what we have done.

We create a folder called Ethereum in the home folder on our Mac called /Users/vijaymukhi/Ethereum. Then we run the following command twice, each time like before, we give the password twice.

```
geth -dev -datadir "/Users/vijaymukhi/Ethereum" account new
```

We have two new options, the -dev makes our life easier, we do not have to specify multiple command line options.

The only problem was that the 3 tutorials given by Google did not work. This includes the Ethereum wiki. The Ethereum developers refuse to update the tutorials.

You can see that the keystore folder has two files that store the keys we just created. The datadir option simply says the Ethereum folder in our home directory contains the blockchain. Nothing else. All our Ethereum data resides here.

```
geth -dev -datadir "/Users/vijaymukhi/Ethereum" -mine
```

Now we are a real miner because we used the mine option.

```
IPC endpoint opened: /Users/vijaymukhi/Ethereum/geth.ipc
```

```
mined potential block      number=9 hash=11e738...b46aca
```

The second line is where we see that blocks are mined. Let this terminal run in the background and we will open a new terminal window.

```
geth attach /Users/vijaymukhi/Ethereum/geth.ipc
```

If we by mistake run `geth attach` as always it will start the real Ethereum network. The IPC endpoint that we use is from the Ethereum folder off the home folder.

at block: 144 (Sun, 22 Oct 2017 19:29:56 IST)

We have also mined 144 blocks.

```
> eth.getBalance(eth.accounts[0])
7860000000000000000000
> eth.getBalance(eth.accounts[1])
0
```

Our coinbase account has lots of useless ether. The second account has none. Let's change this.

```
eth.sendTransaction({from:eth.accounts[0] , to:eth.accounts[1] , value:1000000000000 })
"0x77cc09bed78deae9250cc2533be1f7d9aba7923c761f1cca5427c26a06511585"
eth.getBalance(eth.accounts[1])
1000000000000
```

We use the `sendTransaction` function of the `eth` object to send ether from one ethereum account to the other. The only difference is that the transaction hash is displayed but everything happens in the blink of an eyelid.

We also see this line in the miner output.

Submitted transaction
 fullhash=0x77cc09bed78daee9250cc2533be1f7d9aba7923c761f1cca5427c26a065115
 85 recipient=0xf075E726A3f98BeCa36bE48313f8989baE982Bc8

That's why we need two windows, one to debug. We end here to maintain the flow, see you for more solidity in Chapter 38.

Please take our advice, develop in dev, and then try it out on the testnet.

In my country we have a saying, thou shall be pardoned a 100 murders. After inventing the dev network option, the Ethereum developers are also pardoned a 100 murders, not individually but cumulatively .

Back to Work

In the good old days, geth had a command line option called `--genesis` which is not available anymore. This option was used to initialize a genesis block, so full flexibility on how our blockchain would look like. This `--dev` option also sets the option `max peers` to 0, whose default value is 25. By setting it to 0, we disable the network, which means that we connect to no one and no one connects to us.

We also set the option `--nodiscovery` on which means that automatic peer discovery is turned off. No looking out for other nodes, the earlier option ensured that no one could find us. The minimal gasprice is set to 18000000000, in developer mode it is set to 0. Even though ether in dev mode is free, we pay no gas for mining.

What takes time in mining blockchains is the proof of work. Ethereum plans to use Proof of Stake at some point in time. The testnet uses a pre-configured proof of work and rinkeby uses a proof of authority. The idea is that we need blocks to be mined instantaneously and the difficulty must never increase.

We learnt about the internals of Ethereum by parsing 300 million leveldb records, it is not for the faint hearted. We created a blockchain of say 100 blocks and then understood the contents of each of them. For example, in the keystore folder as we have two accounts, there are two text files. These files start with a key called `address` which is the Ethereum address that we just created. The file `geth.ipc` that we used with `attach` command only exists when geth is running. This file is very special as it starts with a `s`, any file that starts with anything but a `d` or a space means trouble. Sticky or `suid` is for a Linux geek.

CHAPTER 33

Ethereum Leveldb Keys and GOLANG

We started this chapter with some confusion in our mind. The dilemma here was to either stick to the basics of a language we already know, Python or learn a new programming language altogether. Initially, we tried working in our comfort zone and leaned towards Python as we were very familiar with its ins and outs. However, we then realized that the favored language in the Ethereum world is go (called golang on Google), Rust comes in the second place. So, we decided to give it a try, how difficult can it be !!!

The favored Ethereum client is geth and it is written in go. Though the language of our choice is and will always be Python, the only reason we continued with golang was the source code of geth. We have learnt Bitcoins, blockchain technologies after reading the source code. So, to understand the internals of Ethereum, we must understand geth and this program is written in golang. Most of the code in this chapter is picked up from the original source code.

A brief history of golang: GoLang stands for Go, nothing more nothing less. It is developed by the same group of people at AT&T who were present during the development of the C programming language. These programmers now at Google created this language to solve the problems of scale at Google. The best thing about golang is its swiftness in building the Ethereum client, geth.

The First Go Program

```
ch3301.go
package main
import "fmt"
func main() {
    fmt.Printf("Hell %d\n", 420)
}
```

The first line of every program in Go is the name of the package; the code following is part of this package. An executable program must always be part of a package called main.

The word func denotes the start of the function and the name of our function is main. The parameters to the function are within round brackets. Now for the killer: Like Python, visual placement of code counts a lot. The first opening { must be on the same line. If not, watch the errors you get.

The next line has a function Printf from the package fmt. The Printf function has the same syntax as C where %d is replaced by the number following. Before using code from a package, the package must be imported in the program, hence the keyword import is used in the beginning.

To run the program, the command is

```
$ go run ch3301.go
```

Output Hell 420

The smallest go program confirms we are living in hell; especially if you come from any other programming language. Unlike Bitcoins, there are no blk.dat files to hold raw block data in Ethereum. Instead, there is a folder called chaindata, either in the main Ethereum folder or in the geth folder. This is the 100GB folder that contains all the Ethereum transactions to date.

Like Bitcoin, Ethereum also uses the same leveldb database to store all the Ethereum transactions and more important, the state. Therefore, Ethereum is termed as a state machine. Again, like Bitcoin, Ethereum also does not overtly document the key names but mentions them in the source code only.

In the core folder of the Ethereum source, there is a file called database_util.go. This file contains all the key names and the golang code that retrieves their values. Most of the code in this chapter is taken from this file.

The First Leveldb Key - BlockchainVersion

```
ch3302.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    data, _ := db.Get([]byte("BlockchainVersion"), nil)
    fmt.Printf("%v:%d:%T:%d\n", data, data[0], data, len(data))
}
```

Output [3]:3:[uint8:1

The import statement brings in the leveldb package. This package has all the functions to open, read and write leveldb code. These package names are mentioned in the source code.

Generally, it is expected of a programmer to mention the type of a variable before creating or using it. The := in golang sets aside this requirement. There is a var keyword instead.

The OpenFile function takes the leveldb folder name as the first parameter. Copy the original leveldb folder, chaindata to the Desktop. The second parameter is nil as no value is specified. This function OpenFile returns two parameters, the first is the leveldb file handle which will be used to call other leveldb functions. The second parameter is the error object.

We use the _ character to store the error value so no errors will be reported. If a variable name is not given, then an error crops up. An _ indicates no variable name given.

The Get function requires a leveldb key name. The first key name is BlockchainVersion. We would have loved it if we could pass a string, but life is never so easy. The string type is replaced by an array of bytes. The string is given in double quotes. The [] before it denotes an array and it is followed by the type of array, byte.

The Get function returns the value of the key given to it. Once again, there are two return values and the second return value is ignored by using a _. The variables names db and data are taken from the original source code.

To display the output, the %v modifier is used. Golang has data types which use a standard default format to display

values. For instance, the %T gives the type of the variable, in our case the variable data is of type [] uint8 or an array of bytes.

The %v modifier displays all the bytes as an array in [] brackets. The notation, data [0] displays the first byte 3 using the %d modifier. Finally, nearly all types have a len function that returns the size of the data, our array is of size 1.

The key BlockchainVersion unravels the version of the Ethereum blockchain being used, 3.

If things do not work, first give the following command in the same folder.

```
go get github.com/syndtr/goleveldb/leveldb
```

Then run (this is optional)

```
export GOPATH=/Users/vijaymukhi/Dropbox/0finalbitcoinbook
```

Replace our folder name with your folder name. Repeat go get for every error.

One More Key - LastBlock

```
ch3303.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    data, _ := db.Get([]byte("LastBlock"), nil)
    fmt.Printf("%02x:%T:%d\n", data, data, len(data) )
}
```

Output

```
8785106269eada467d1d94cb8e13b067609752a32f87aca0b87eeb05ae9391cb:[]uint8:32
```

This example reads the key called LastBlock. The value of this key is not a block number but the hash value of the last mined block. The value returned is an array of bytes, 32 large.

The Ethereum client uses this approach to download the next block on our machine. In the Ethereum Browser, paste this hash value. The output window reveals that there is no transaction hash by this name.

Thus, even though the block hash values are stored in the leveldb folder, the Ethereum explorer refrains from displaying them. Your mileage in most code will vary as we have different leveldb databases.

Two More Keys : LastHeader and LastFast

```
ch3304.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
)
func main() {
```



```

db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
data, _ := db.Get([]byte("LastBlock"), nil)
fmt.Printf("%02x\n", data)
data, _ = db.Get([]byte("LastHeader"), nil)
fmt.Printf("%02x\n", data)
data, _ = db.Get([]byte("LastFast"), nil)
fmt.Printf("%02x\n", data)
}

```

Output

```

8785106269eada467d1d94cb8e13b067609752a32f87aca0b87eeb05ae9391cb
8785106269eada467d1d94cb8e13b067609752a32f87aca0b87eeb05ae9391cb
8785106269eada467d1d94cb8e13b067609752a32f87aca0b87eeb05ae9391cb

```

The program displays two more keys, all three, LastHeader and LastFast and LastBlock give the same hash value. In Byzantium, the three keys have different values, our blockchain was not in sync.

A Key h <block number> n to Retrieve a Block Hash

```

ch3305.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    number := make([]byte, 8)
    binary.BigEndian.PutUint64(number, uint64(300))
    fmt.Printf("%v:%T\n", number, number)
    key := append([]byte("h"), number...)
    fmt.Printf("%v:%d\n", key, len(key))
    key = append(key, []byte("n")...)
    fmt.Printf("%v:%d\n", key, len(key))
    data, _ := db.Get(key, nil)
    fmt.Printf("%x:%T:%v\n", data, data, data)
}

```

Output

```

[0 0 0 0 0 1 44]:[]uint8
[104 0 0 0 0 0 1 44]:9
[104 0 0 0 0 0 1 44 110]:10
b3e37f7c14742bc54d08163792d38ada69c3951817b8dde6ef96776aa5c0f00c:[]uint8:[
179 227 127 124 20 116 43 197 77 8 22 55 146 211 138 218 105 195 149 24 23 184
221 230 239 150 119 106 165 192 240 12]

```

The first hurdle is dealing with numbers. In the program, we want to obtain the block hash value of block number 300.

For this task, the number must be part and parcel of a long key name or string or array of bytes. The number 300 in bytes would be 44 and 1 or 1 and 44 depending upon the endianness.

First, using the make function, an array is created of type bytes with a size value of 8. The make function is very versatile, it needs a type and a size value. Then the BigEndian function does its job. It takes a number as the second parameter and writes it out as a series of 8 bytes in the number array. The uint64 function is not necessary. At the end of this function, the variable number has a series of 6 0's followed by a 1 and a 44. The array size is 8 bytes. There is a function called LittleEndian also. The value is stored in a string as an array of bytes.

Now, a lower-case h is added to the beginning of the byte array number. The append function takes the string h or the ascii value of 104 and adds it to the number array. Ignore the triple dots for the moment. The array number is now 9 bytes large starting with a 104. Basically, append adds the first parameter array to the second parameter array.

Now for the reverse. The character n must be added at the end of the array key. So, the array named key is given as the first parameter and the string n as the second parameter. The array is now 10 large, it ends with a 110, the ascii value of n.

The resultant value is a hash of the block 300. It starts with b3e3. The Ethereum blockchain explorer endorses this hash value for block number 300.

The explorer assumes that all hashes given to it are transaction hashes. A little later, we do the reverse, given a hash you can get the block number.

This go lang function, Printf has plenty of inbuilt options to display a hash or byte array in different formats. A %x modifier gives a hash as a series of hex bytes. A %v modifier shows the same bytes as decimal numbers, like a normal array.

In Byzantium, we obviously get the same answer as the block number is the same.

A Key h <number> <hash> to Retrieve all the Block Details

```
ch3306.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/core/types"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    number := make([]byte, 8)
    binary.BigEndian.PutUint64(number, uint64(300))
    key := append([]byte("h"), number...)
    key = append(key, []byte("n")...)
    fmt.Printf("First Key %s:%d:%s\n", "h", number, "n")
    hash, _ := db.Get(key, nil)
    fmt.Printf("%x\n", hash)
    key1 := append(append([]byte("h"), number...), hash...)
    fmt.Printf("Second Key %s:%d:%x\n", "h", number, hash)
```

}

Output

[

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
Difficulty: 19753900789
Number: 300
GasLimit: 5000
GasUsed: 0
Time: 1438270848
Extra: Geth/LVIV/v1.0.0/linux/go1.4.2
MixDigest: 809d462db719691e7914720e2cda25994a98b1b71402ca11eac9224b68cb2cea
Nonce: 424b554fa4a7a04f
```

]

A fair warning, if the code is beyond your understanding, please read the explanations again and then move on to the next example where the go-lang code is further simplified.

First, the hash value of block number 300 is obtained and it is assigned to a variable, `hash`. Now, a key is created, a hideous one. It starts with a `h`, followed by an 8-byte block number and finally the hash value of the block number. Here the block number and the hash value is of block number 300. The key is stored in variable `key1` and its value is displayed.

When the value of this key is retrieved, it is 539 bytes large. We have not shown the entire value as it looks like some random bytes.

Then a variable called `header` is created with a `Header` data type. This data type is in the `types` package. It can be an in-built structure or class in Ethereum. This code is present in the folder `go-ethereum/core/types`. The modifier `%T` confirms the data type of the variable `header`. The new function creates types on the fly.

The `%v` modifier displays the value of this variable, `header` in the required format. The output shows that every field of this structure is initialized to 0. You can see the name of every field and its value.

The value of the key stored in the variable `data` is read using the `NewReader` function of the `byte's` class. The type of the variable `tmp` is of type `bytes.Reader`. Once the bytes are read into a `Reader` object `tmp`, the `Decode` function of the `rlp` package is used to decode the data. This function uses the type of the `header` variable to determine the structure of the data. The initial bytes when displayed start with `f9`.

Ethereum invented its own encoding called Recursive Length Prefix or RLP. The `Decode` function takes the reader variable type `tmp` and then decodes the bytes into the `header` variable.

The modifier `%v` is used to display the `header` variable. The output is similar to the one shown by Geth. After all, it is the same code base down to variable names. Do note that the fieldnames in the `Header` are not chosen by us.

Use the block number instead of the block hash in your favorite Ethereum blockchain explorer and see the output.

In `geth`, enter the following command

```
eth.getBlock(300)
{
  difficulty: 19753900789,
  extraData: "0x476574682f4c5649562f76312e302e302f6c696e75782f676f312e342e32",
  gasLimit: 5000,
  gasUsed: 0,
```

```

hash: "0xb3e37f7c14742bc54d08163792d38ada69c3951817b8dde6ef96776aa5c0f00c",
logsBloom:
"0x0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000",
miner: "0xbb7b8287f3f0a933474a79eae42cbca977791171",
mixHash:
"0x809d462db719691e7914720e2cda25994a98b1b71402ca11eac9224b68cb2cea",
nonce: "0x424b554fa4a7a04f",
number: 300,
parentHash:
"0x989b8bf2af0be6c18c9c95bfde81492e0b47bcc1c26d555bb7cea2d09e92c6c3",
receiptsRoot:
"0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cad001622fb5e363b421",
sha3Uncles:
"0x1dcc4de8dec75d7aab85b567b6ccdd41ad312451b948a7413f0a142fd40d49347",
size: 544,
stateRoot:
"0x34e5b52497408cd2bbcb6992dee0292498a235ec7aca1b34f6cbccb396f85105",
timestamp: 1438270848,
totalDifficulty: 5531721283386,
transactions: [],
transactionsRoot:
"0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cad001622fb5e363b421",
uncles: []
}

```

This output re-confirms that code remains the same irrespective of who uses it. There is no reason for Byzantium to give us a different answer.

A Simplified version of the earlier program

```

ch3307.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    //"github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/core/types"

```

```
        "github.com/ethereum/go-ethereum/common"
    )
    type Header1 struct {
        ParentHash common.Hash
        UncleHash common.Hash
    }
    func (h *Header1) String() string {
        return fmt.Sprintf("Vijay Mukhi:
        [
            ParentHash:    %x
            UncleHash:     %x
        ]", h.ParentHash, h.UncleHash)
    }
    func main() {
        db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
        number := make([]byte, 8)
        binary.BigEndian.PutUint64(number, uint64(300))
        key := append([]byte("h"), number...)
        key = append(key, []byte("n")...)
        hash, _ := db.Get(key, nil)
        key1 := append(append([]byte("h"), number...), hash...)
        data, _ := db.Get(key1, nil)
        header := new(types.Header)
        rlp.Decode(bytes.NewReader(data), header)
        fmt.Printf("%+v\n", header)
        header1 := new(Header1)
        rlp.Decode(bytes.NewReader(data), header1)
        fmt.Printf("%+v\n", header1)
        fmt.Printf("%x:%x\n", header1.ParentHash, header1.UncleHash)
    }
}
```

Output

Header(b3e37f7c14742bc54d08163792d38ada69c3951817b8dde6ef96776aa5c0f00c):

```
[
    ParentHash:
    989b8bf2af0be6c18c9c95bfde81492e0b47bcc1c26d555bb7cea2d09e92c6c3
    UncleHash:
    1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
    Coinbase:
    bb7b8287f3f0a933474a79eae42cbca977791171
    Root:
    34e5b52497408cd2bbcb6992dee0292498a235ec7aca1b34f6cbccb396f85105
    TxSha
    56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cad001622fb5e363b421
    ReceiptSha:
    56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cad001622fb5e363b421
]
```



```
ch3308.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/core/types"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    number := make([]byte, 8)
    binary.BigEndian.PutUint64(number, uint64(1000004)) // 1000004 no uncles but
    few transactions 2586550 one uncle
    key := append([]byte("h"), number...)
    key = append(key, []byte("n")...)
    hash, _ := db.Get(key, nil)
    key1 := append(append([]byte("b"), number...), hash...)
    fmt.Printf("%v\n", key1)
    data, _ := db.Get(key1, nil)
    fmt.Printf("Data Size %d\n", len(data))

    body := new(types.Body)
    rlp.Decode(bytes.NewReader(data), body)
    fmt.Printf("%+v\n", body)
}
```

Output

```
[98 0 0 0 0 15 66 68 92 38 137 210 123 254 222 217 250 160 213 46 115 1 187 66
94 7 88 238 43 85 11 133 37 87 119 110 84 83 237 72]
Data Size 232
&{Transactions:[
    TX(4cec043eca5a4b936fbf911fd3be02e0bb1b093c2882bde37ea08506442fb889)
    Contract: false
    From: 120a270bbc009644e35f0bb6ab13f95b8199c4ad
    To: 3dc12a32a5abf477e2ec91f6218d0a96150fef99
    Nonce: 12555
    GasPrice: 50000000000
    GasLimit 250000
    Value: 1048847840000000000
    Data: 0x
    V: 0x1c
    R:
    0x4e64af986b1e5dd2413d46a735c5ff6769bae4dd4ae9274e0cc4b8ba8c5d16e5
    S:
    0x49bdb44f735f0d2c43e09a7c86396e0192d0dd4f0da54ce7fd8d8fe8ef8861eb
```



```

Hex:
f86f82310b850ba43b74008303d090943dc12a32a5abf477e2ec91f6218d0a96150fef998
80e8e418daea2c000801ca04e64af986b1e5dd2413d46a735c5ff6769bae4dd4ae9274e0
cc4b8ba8c5d16e5a049bdb44f735f0d2c43e09a7c86396e0192d0dd4f0da54ce7fd8d8fe
8ef8861eb
TX(0baaaf107ec45688f429be367189890e220d5020e17f64105fd76bef894e8fc6)
Contract: false
From: 2a65aca4d5fc5b5c859090a6c34d164135398226
To: f4f2c15602b084cae84ea603f75527de19705aa1
Nonce: 172324
GasPrice: 50000000000
GasLimit 90000
Value: 10414606500000000000
Data: 0x
V: 0x1c
R:
0xac560d0a391f023603772db41c8d2073e4ab43d8bf8f320b94b4b9d2e6a46a68
S:
0xdf2ae610522564553f8851905e8289192cea616a550dee5bd787fe3d296912d
Hex:
f8708302a124850ba43b740083015f9094f4f2c15602b084cae84ea603f75527de197
05aa1880e7402f17acc2400801ca0ac560d0a391f023603772db41c8d2073e
4ab43d8bf8f320b94b4b9d2e6a46a68a00df2ae610522564553f8851
905e8289192cea616a550dee5bd787fe3d296912d
] Uncles:[]

```

As before, the hash of block number 1000004 is saved in the hash variable.

Then, b or ASCII value 98 is added, followed by the block number and then the hash of the block. There are, in all, 232 bytes of data. A new variable of type Body is created and the decoded bytes are stored in the data variable.

The block number 100004 has two transactions and they are displayed. No extra work from our side. Our blockchain explorer also confirms the same.

There is a header followed by a body.

```
binary.BigEndian.PutUint64(number , uint64(2586550))
```

We make only one change. The block number is 2586550.

Output

```

Uncles:[Header(70a9519b5aa35e2b04176bc365fbbfd31198aeaf115b25bd6884987d9943df67):
[
  ParentHash:
  00aaf53b5a0cdad19a11ceb0a9a2454151e9a6045d17a75d7f865fddcc81644c
  UncleHash:
  1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
  Coinbase:
  4bb96091ee9d802ed039c4d1a5f6216f90f81b01

```

113

A Key h <block number> Hash t

738

```

    fmt.Printf("%v\n", key1)
    data, _ := db.Get(key1, nil)
    td := new(big.Int)
    rlp.Decode(bytes.NewReader(data), td)
    fmt.Printf("%+v\n", td)
}

```

Output

```

[104 0 0 0 0 0 15 66 68 92 38 137 210 123 254 222 217 250 160 213 46 115 1 187 66
94 7 88 238 43 85 11 133 37 87 119 110 84 83 237 72 116]
+7135252673908215265

```

This key is the one of the most complex one. It starts with the letter h, then the block number 1000004 followed by the block hash and ends with a t. The return value is the total difficulty of the block. This difficulty value has a special key name reserved per block. This time we will not use the Byzantium word.

Key Block Hash H

```

ch3310.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    number := make([]byte, 8)
    binary.BigEndian.PutUint64(number, uint64(300))
    key := append([]byte("h"), number...)
    key = append(key, []byte("n")...)
    fmt.Printf("%v\n", key)
    hash, _ := db.Get(key, nil)
    fmt.Printf("%x\n", hash)
    key1 := append([]byte("H"), hash...)
    data, _ := db.Get(key1, nil)
    fmt.Printf("%+v\n", data)
    blockno := binary.BigEndian.Uint64(data)
    fmt.Printf("Block Number %d:%T\n", blockno, blockno)
}

```

Output

```

[104 0 0 0 0 0 1 44 110]
b3e37f7c14742bc54d08163792d38ada69c3951817b8dde6ef96776aa5c0f00c
[0 0 0 0 0 1 44]
Block Number 300:uint64

```

This one squares the circle. The variable key starts with the h, then the block number followed by a n. The block hash as before is assigned to the hash variable.

The purpose of this program is to obtain the block number of this hash value. The blockchain explorer is of no help here. Simply add an H to the hash. The Get function now returns the block number but as a series of bytes. The package binary helps convert the BigEndian number into a simple integer. The same block number, 300 is shown. Passes Byzantium.

-Receipts Key r <block number> Hash

```
ch3311.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
    "github.com/ethereum/go-ethereum/rlp"
    "github.com/ethereum/go-ethereum/core/types"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    number := make([]byte, 8)
    binary.BigEndian.PutUint64(number, uint64(2000004))
    key := append([]byte("h"), number...)
    key = append(key, []byte("n")...)
    hash, _ := db.Get(key, nil)
    key1 := append(append([]byte("r"), number...), hash...)
    data, _ := db.Get(key1, nil)
    fmt.Printf("data length %d\n", len(data))
    storageReceipts := []*types.ReceiptForStorage{}
    fmt.Printf("Type %T\n", storageReceipts)
    rlp.DecodeBytes(data, &storageReceipts)
    fmt.Printf("Length %d\n", len(storageReceipts))
    receipts := make(types.Receipts, len(storageReceipts))
    fmt.Printf("Length %d:%T\n", len(receipts), receipts)
    for i, receipt := range storageReceipts {
        fmt.Printf("Type %T\n", receipt)
        receipts[i] = (*types.Receipt)(receipt)
        fmt.Printf("(%d)%+v\n", i, receipts[i])
        fmt.Printf("%x:%x:%d\n", receipts[i].TxHash, receipts[i].ContractAddress,
            receipts[i].GasUsed)
    }
}
```

Output

```
data length 1628
Type []*types.ReceiptForStorage
Length 4
Length 4:types.Receipts
Type *types.ReceiptForStorage
```

```
(0)receipt{med=56e181c14e3d6834615336a6c1e335dbded1f19e4fa63f12468d333144
067d7c cgas=21000
bloom=0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
000000000000 logs=[]}
4de721391f9075bc0d5c27d09569dcba8975d78258ed527a6d287474a087bd34:000000
0000000000000000000000000000000000000000000000000000000000000000:21000
```

In this program, the hash value of block number 2000004 is retrieved and saved in the hash variable. The key now starts with an r, then the block number and finally the block hash value. The value returned for this key is very large. The data size is a total of 1628 bytes.

An array of ReceiptStorage objects is defined in a variable called storageReceipts. As the value received is rlp encoded, the DecodeBytes function is used to decode the rlp encoded data into the storageReceipts variable. Our value consists of 4 ReceiptForStorage objects.

Now let's display the members of the array. The make function is used to create an array of 4 Receipts objects.

The for loop in go lang takes 2 variables. First is the index variable i and second one is the receipt variable. This receipt variable is not an array but a simple ReceiptForStorage type. The storageReceipts accesses each member of the array.

The operator () is used to cast the ReceiptForStorage type to a Receipt type. Then in the for loop, variable i is used to initialize each array member. The members are initialized using the [] operator.

Finally, the %v modifier in the Printf function shows the receipt fields, like med and logs. Other members of the receipts type like TxHash, GasUsed etc. are also displayed. One of the four Receipt objects is shown and it should suffice for now.

Reading a Key that Represents a Transaction

```
ch3312.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/common"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    txhash :=
    []byte{0x4c, 0xec, 0x04, 0x3e, 0xca, 0x5a, 0x4b, 0x93, 0x6f, 0xbf, 0x91, 0x1f, 0xd3, 0xbe,
    0x02, 0xe0, 0xbb, 0x1b, 0x09, 0x3c, 0x28, 0x82, 0xbd, 0xe3, 0x7e, 0xa0, 0x85, 0x06, 0x44,
    0x2f, 0xb8, 0x89}
```

```
txhash :=
common.HexToHash("4cec043eca5a4b936fbf911fd3be02e0bb1b093c2882bde37ea08506442fb889")
datat, _ := db.Get(txhash.Bytes(), nil)
fmt.Printf("%x\n", txhash.Bytes())
tx := new(types.Transaction)
rlp.Decode(bytes.NewReader(datat), tx)
fmt.Printf("%+v\n", tx)
}
```

Output

```
4cec043eca5a4b936fbf911fd3be02e0bb1b093c2882bde37ea08506442fb889
TX(4cec043eca5a4b936fbf911fd3be02e0bb1b093c2882bde37ea08506442fb889)
Contract: false
From: 120a270bbcb009644e35f0bb6ab13f95b8199c4ad
To: 3dc12a32a5abf477e2ec91f6218d0a96150fef99
Nonce: 12555
GasPrice: 50000000000
GasLimit 250000
Value: 1048847840000000000
Data: 0x
V: 0x1c
R:
0x4e64af986b1e5dd2413d46a735c5ff6769bae4dd4ae9274e0cc4b8ba8c5d16e5
S:
0x49bdb44f735f0d2c43e09a7c86396e0192d0dd4f0da54ce7fd8d8fe8ef8861eb
Hex:
f86f82310b850ba43b74008303d090943dc12a32a5abf477e2ec91f6218d0a96150fef998
80e8e418daea2c000801ca04e64af986b1e5dd2413d46a735c5ff6769bae4dd4ae9274e0
cc4b8ba8c5d16e5a049bdb44f735f0d2c43e09a7c86396e0192d0dd4f0da54ce7fd8d8fe
8ef8861eb
```

A transaction is the heart and soul of any blockchain. No transactions -> no blockchain. A key by itself, with no characters before or after, stands for a transaction hash.

We have deciphered a certain transaction hash in the program. The function HexToHash converts the hex bytes to a string, as in an array of bytes. The byte array can be created manually using the individual bytes as well. The Transaction encoded object is decoded and then displayed. Once again, no effort in learning what the individual field names are.

In Byzantium, we get no error but no useful output. For some reason, the transaction hash is nonexistent in the blockchain explorer. We had no idea why the hash displayed is not the hash we specified.

```
TX(c5b2c658f5fa236c598a6e7fbf7f21413dc42e2a41dd982eb772b30707cba2eb)
```

We then read the comments in the file database_util.go and this is what we see.

```
// Old transaction representation, load the transaction and it's metadata separately
```

This could only mean that the transaction data is stored differently in Byzantium. As transactions are the backbone of Ethereum, we have added a program at the very end.


```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
000000000000000000] Logs:[] TxHash:[76 236 4 62 202 90 75 147 111 191 145
31 211 190 2 224 187 27 9 60 40 130 189 227 126 160 133 6 68 47 184 137]
ontractAddress: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] GasUsed:+21000}
```

The same transaction hash value is now used as individual bytes, for no apparent reason. Then the string receipts are added to the beginning of the key name. In all, 356 bytes of data are returned. Now, the official var keyword is used to create a variable called receipt of type ReceiptForStorage. Thereafter, the function DecodeBytes decodes the rlp encoded data in the datar variable.

The same combination of new and Decode produces the same result. Both variables, receipt and receipt1 are displayed and they have the same value. Like the C language, the & in golang indicates there are pointers as well. There are many ways to skin a cat. Along the way, we are learning many new features of golang.

In Byzantium, all 0's. Once again, a comment to the rescue.

```
// used by old db, now only used for conversion
```

This comment is found while defining an array of bytes representing the key receipts-. As this is an old key, we are in no mood to do the new key which is similar to handling the new transaction data.

A Transaction Hash With a 01 Added to the Very End

```
ch3314.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/common"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    txhash :=
    []byte{0x0b,0xaa,0xaf,0x10,0x7e,0xc4,0x56,0x88,0xf4,0x29,0xbe,0x36,0x71,0x89,
    0x89,0x0e,0x22,0x0d,0x50,0x20,0xe1,0x7f,0x64,0x10,0x5f,0xd7,0x6b,0xef,0x89,
    0x4e,0x8f,0xc6}
    datat, _ := db.Get(txhash, nil)
    fmt.Printf("%x\n", txhash)
    tx := new(types.Transaction)
    rlp.Decode(bytes.NewReader(datat), tx)
    fmt.Printf("%+v\n", tx)
    key := append(txhash, []byte{0x01}...)
    fmt.Printf("%x\n", key)
    data, _ := db.Get(key, nil)
    fmt.Printf("data %d\n", len(data))
}
```



```

var meta struct {
    BlockHash common.Hash
    BlockIndex uint64
    Index uint64
}
rlp.DecodeBytes(data, &meta)
fmt.Printf("BlockHash %x Block Index %d\n", meta.BlockHash, meta.BlockIndex)
fmt.Printf("%d\n", meta.Index)
}

```

Output

```

0baaaf107ec45688f429be367189890e220d5020e17f64105fd76bef894e8fc6
TX(0baaaf107ec45688f429be367189890e220d5020e17f64105fd76bef894e8fc6)
Contract: false
From: 2a65aca4d5fc5b5c859090a6c34d164135398226
To: f4f2c15602b084cae84ea603f75527de19705aa1
Nonce: 172324
GasPrice: 50000000000
GasLimit 90000
Value: 10414606500000000000
Data: 0x
V: 0x1c
R:
0xac560d0a391f023603772db41c8d2073e4ab43d8bf8f320b94b4b9d2e6a46a68
S:
0xdf2ae610522564553f8851905e8289192cea616a550dee5bd787fe3d296912d
Hex:
f8708302a124850ba43b740083015f9094f4f2c15602b084cae84ea603f75527de19705aa
1880e7402f17acc2400801ca0ac560d0a391f023603772db41c8d2073e4ab43d8bf8f320
b94b4b9d2e6a46a68a00df2ae610522564553f8851905e8289192cea616a550dee5bd78
7fe3d296912d
0baaaf107ec45688f429be367189890e220d5020e17f64105fd76bef894e8fc601
data 39
BlockHash 5c2689d27bfed9faa0d52e7301bb425e0758ee2b550b852557776e5453ed48 Block
Index 1000004
1

```

The transaction hash value starts with 0baa. All the transaction data is read into a variable `data`. Next, a transaction type `tx` is created and the transaction object is decoded and displayed. A number, 01 is added at the end of the transaction hash. Once the key is generated, we ask for the value of this key.

The return value is 39 bytes of data. This data represents a structure or type called `meta`.

The `meta` type gives the hash value of the block it resides in. The block index number is also obtained. Finally, the physical index of the transaction in the list of transactions is displayed.

The explorer must show the same result.

For the same reasons that we mentioned above, transactions do not work.

A Key Called ethereum-config

```
ch3315.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
    "encoding/json"
    "math/big"
    "github.com/ethereum/go-ethereum/common"
)
type ChainConfig struct {
    ChainId *big.Int 'json:"chainId"' // Chain id identifies the current chain and i
    s used for replay protection

    HomesteadBlock *big.Int 'json:"homesteadBlock"' // Homestead switch
    block (nil = no fork, 0 = already homestead)
    DAOForkBlock *big.Int 'json:"daoForkBlock"' // TheDAO hard-fork
    switch block (nil = no fork)
    DAOForkSupport bool 'json:"daoForkSupport"' // Whether the nodes
    supports or opposes the DAO hard-fork
    // EIP150 implements the Gas price changes
    (https://github.com/ethereum/EIPs/issues/150)
    EIP150Block *big.Int 'json:"eip150Block"' // EIP150 HF block (nil = no
    fork)
    EIP150Hash common.Hash 'json:"eip150Hash"' // EIP150 HF hash (fast
    sync aid)
    EIP155Block *big.Int 'json:"eip155Block"' // EIP155 HF block
    EIP158Block *big.Int 'json:"eip158Block"' // EIP158 HF block
}
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    number := make([]byte, 8)
    binary.BigEndian.PutUint64(number, uint64(0))
    key := append([]byte("h"), number...)
    key = append(key, []byte("n")...)
    fmt.Printf("%v\n", key)
    hash, _ := db.Get(key, nil)
    fmt.Printf("%x\n", hash)
    key1 := append(append([]byte("ethereum-config-"), hash...))
    jsonChainConfig, _ := db.Get(key1, nil)
    var config ChainConfig
    json.Unmarshal(jsonChainConfig, &config)
    fmt.Printf("%+v\n", config)
}
```

Output

```
[104 0 0 0 0 0 0 0 110]
d4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3
{ChainId:+1 HomesteadBlock:+1150000 DAOForkBlock:+1920000
DAOForkSupport:true EIP150Block:+2463000 EIP150Hash:[32 134 121 154 238 190
174 19 92 36 108 101 2 28 130 180 225 90 44 69 19 64 153 58 172 253 39 81 136 101
20 240] EIP155Block:+2675000 EIP158Block:+2675000}
```

First, a key that starts with a h is created. It is followed by 8 0's (the block number) and ends with a n. This is the hash of block 0. Yes, everyone has a genesis block.

Then, the string ethereum-config- is appended to this genesis hash value. We are returned a type, ChainConfig. Source is the source code. This value is not rlp encoded like the others. The function in the package json called Unmarshal decodes or unmarshals the value. These are basically block numbers when forks took place in the Ethereum network. The DAO fork needs a separate book as it nearly broke Ethereum for no fault of theirs.

The Last Key Setting-Mipmap-Version

```
ch3316.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    //"fmt"
    "fmt"
    "github.com/ethereum/go-ethereum/common"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    key := common.Hex2Bytes("7365747469726469706461707264697273696665")
    data, _ := db.Get(key, nil)
    fmt.Printf("%x:%d\n", data, len(data))
    fmt.Printf("%s\n", key)
}
```

Output

```
02:1
setting-mipmap-version
```

The key is written in hex bytes but it is a simple string, setting-mipmap-version. You can use the old method of converting a string into hex bytes.

The output representing the version is a single byte, 2.

This time in Byzantium, the version is :0 and the length of data is 0. This means that the key simply does not exist.

The new transaction leveldb key and value.

```
Ch3317.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
```

```
    "fmt"
    "encoding/binary"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/common"
)
type txLookupEntry struct {
    BlockHash common.Hash
    BlockIndex uint64
    Index uint64
}
func encodeBlockNumber(number uint64) []byte {
    enc := make([]byte, 8)
    binary.BigEndian.PutUint64(enc, number)
    return enc
}
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    txhash :=
    common.HexToHash("4cec043eca5a4b936fbf911fd3be02e0bb1b093c2882bde37ea08
    506442fb889")
    var lookupPrefix = []byte("l")
    data, _ := db.Get(append(lookupPrefix, txhash.Bytes()...), nil)
    var entry txLookupEntry
    rlp.DecodeBytes(data, &entry)
    blockHash := entry.BlockHash
    blockNumber := entry.BlockIndex
    txIndex := entry.Index
    fmt.Printf("0x%x:0x%x:0x%x\n", blockHash, blockNumber, txIndex)
    var bodyPrefix = []byte("b")
    aa := append(append(bodyPrefix, encodeBlockNumber(blockNumber)...),
    blockHash.Bytes()...)
    data1, _ := db.Get(aa, nil)
    body := new(types.Body)
    rlp.Decode(bytes.NewReader(data1), body)
    fmt.Printf("%+v\n", body)
    datat := body.Transactions[txIndex]
    fmt.Printf("%+v\n", datat)
}
```

Output

```
5c2689d27bfed9faa0d52e7301bb425e0758ee2b550b852557776e5453ed48:100000
4:0
&{Transactions:[
    TX(4cec043eca5a4b936fbf911fd3be02e0bb1b093c2882bde37ea08506442fb889)
    Contract: false
```

```

From: 120a270bbc009644e35f0bb6ab13f95b8199c4ad
To: 3dc12a32a5abf477e2ec91f6218d0a96150fef99
Nonce: 12555
GasPrice: 0xba43b7400
GasLimit 0x3d090
Value: 0xe8e418daea2c000
Data: 0x
V: 0x1c
R:
0x4e64af986b1e5dd2413d46a735c5ff6769bae4dd4ae9274e0cc4b8ba8c5d16e5
S:
0x49bdb44f735f0d2c43e09a7c86396e0192d0dd4f0da54ce7fd8d8fe8ef8861eb
Hex:
f86f82310b850ba43b74008303d090943dc12a32a5abf477e2ec91f6218d0a96150fef998
80e8e418daea2c000801ca04e64af986b1e5dd2413d46a735c5ff6769bae4dd4ae9274e0
cc4b8ba8c5d16e5a049bdb44f735f0d2c43e09a7c86396e0192d0dd4f0da54ce7fd8d8fe
8ef8861eb
TX(0baaaf107ec45688f429be367189890e220d5020e17f64105fd76bef894e8fc6)
Contract: false
From: 2a65aca4d5fc5b5c859090a6c34d164135398226
To: f4f2c15602b084cae84ea603f75527de19705aa1
Nonce: 172324
GasPrice: 0xba43b7400
GasLimit 0x15f90
Value: 0xe7402f17acc2400
Data: 0x
V: 0x1c
R:
0xac560d0a391f023603772db41c8d2073e4ab43d8bf8f320b94b4b9d2e6a46a68
S:
0xdf2ae610522564553f8851905e8289192cea616a550dee5bd787fe3d296912d
Hex:
f8708302a124850ba43b740083015f9094f4f2c15602b084cae84ea603f75527de19705aa
1880e7402f17acc2400801ca0ac560d0a391f023603772db41c8d2073e4ab43d8bf8f320
b94b4b9d2e6a46a68a00df2ae610522564553f8851905e8289192cea616a550dee5bd78
7fe3d296912d
] Uncles:[]
TX(4cec043eca5a4b936fbf911fd3be02e0bb1b093c2882bde37ea08506442fb889)
Contract: false
From: 120a270bbc009644e35f0bb6ab13f95b8199c4ad
To: 3dc12a32a5abf477e2ec91f6218d0a96150fef99
Nonce: 12555
GasPrice: 0xba43b7400
GasLimit 0x3d090
Value: 0xe8e418daea2c000
Data: 0x

```

```
V: 0x1c
R:
0x4e64af986b1e5dd2413d46a735c5ff6769bae4dd4ae9274e0cc4b8ba8c5d16e5
S:
0x49bdbb44f735f0d2c43e09a7c86396e0192d0dd4f0da54ce7fd8d8fe8ef8861eb
Hex:
f86f82310b850ba43b74008303d090943dc12a32a5abf477e2ec91f6218d0a96150fef998
80e8e418daea2c000801ca04e64af986b1e5dd2413d46a735c5ff6769bae4dd4ae9274e0
cc4b8ba8c5d16e5a049bdbb44f735f0d2c43e09a7c86396e0192d0dd4f0da54ce7fd8d8fe
8ef8861eb
```

The level db key now starts with the letter l and followed by the transaction hash. The value of the key is now a structure that we create called txLookupEntry. We borrowed this code from the source in the file database_util.go. The DecodeBytes function as always, gives us three different values.

The value starts with a 32-byte block hash this transaction resides in. This is followed by the block number. Finally, an index into the number of transactions this block contains which is 1 in our case. Our favorite block chain explorer confirms that this block number 1000004 has two transactions only and we can confirm the transaction hashes.

Now for the body of the block. We create a key starting with a b and then adding the block number and then the block hash. Either do it our way or bring in a function called encodeBlockNumber from the source code.

We then display the entire body of the two transactions or simply the second transaction, your call.

Counting the Key-Value Pairs in the State

```
ch3317.py
import leveldb
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chaindata")
cnt = 0
for k, v in db.RangeIter():
    cnt = cnt + 1
print "Total number of key value pairs %d" % cnt
```

Output

Total number of key value pairs 197,421,958

The Python code counts the number of key-value pairs in the Ethereum state.

The number of key-value pairs in the chaindata folder is displayed. This beats Bitcoin hands down.

This program takes too long. By the time you run this, the number of key value pairs would have crossed 200 million.

A Count of Every Key in the Ethereum State Machine

```
ch3318.py
import leveldb
db = leveldb.LevelDB("/Users/vijaymukhi/Library/Ethereum/geth/chaindata")
cnt = 0
cntver = 0
cntlastblk = 0
cntlastfast = 0
```

```

cntlastheader = 0
cnthnumbern = 0
cnthnumberhash = 0
cnthnumberhasht = 0
cntHhash = 0
cntbnumberhash = 0
cnt32 = 0
cnthash1 = 0
cntrnumberhash = 0
cntreceiptshash = 0
cntethereum = 0
cntminimap = 0
cntextra = 0
cntsecurekey = 0
cntlog = 0
cntBloomBits = 0
cntBloomBitsIndexPrefix = 0
for k , v in db.Rangelter():
    cnt = cnt + 1
    if cnt % 1000000 == 0:
        print cnt
    if k == "BlockchainVersion" :
        cntver = cntver + 1
    elif k == "LastBlock" :
        cntlastblk = cntlastblk + 1
    elif k == "LastHeader" :
        cntlastheader = cntlastheader + 1
    elif k == "LastFast" :
        cntlastfast = cntlastfast + 1
    elif len(k) == len('ethereum-config-') + 32 :
        cntethereum = cntethereum + 1
    elif k == 'setting-mipmap-version':
        cntminimap = cntminimap + 1
    elif ord(k[0]) == 72 and len(k) == 33:
        cntHhash = cntHhash + 1
    elif len(k) == 32:
        cnt32 = cnt32 + 1
    #print k.encode('hex')
    #print v.encode('hex')
    #exit()
    elif len(k) == 33 :
        cnthash1 = cnthash1 + 1
    # print "%s:%c:%d" % (k.encode('hex') , ord(k[0]) , ord(k[0]))
    # print v.encode('hex')
    # exit()
    elif ord(k[0]) == 98 :
        if len(k) == 41:

```

```
cntbnumberhash = cntbnumberhash + 1
else:
print "A1 %d:%d %d:%c" % (len(k) , len(v) , ord(k[0]) , ord(k[0]))
print len(k)
exit(0)
elif ord(k[0]) == 104 : #checking for a n<number>h a total of a fixed 10 bytes
if len(k) == 10:
cnthnumbern = cnthnumbern + 1
elif len(k) == 41:
cnthnumberhash = cnthnumberhash+ 1
elif len(k) == 42 and ord(k[len(k) -1]) == 116:
cnthnumberhasht = cnthnumberhasht + 1
else :
print "A2 %d:%d %d:%c" % (len(k) , len(v) , ord(k[0]) , ord(k[0]))
#exit(0)
elif ord(k[0]) == 114 and ord(k[1]) == 101 and ord(k[2]) == 99 :
cntreceiptshash = cntreceiptshash + 1
elif ord(k[0]) == 114 and len(k) == 41:
cntrnumberhash = cntrnumberhash + 1
elif k[0:11] == "secure-key-" :
cntsecurekey = cntsecurekey + 1
elif k[0:17] == "mipmap-log-bloom-" :
cntlog = cntlog + 1
elif ord(k[0]) == 66:
cntBloomBits = cntBloomBits + 1
elif ord(k[0]) == 105 and ord(k[1]) == 66:
cntBloomBitsIndexPrefix = cntBloomBitsIndexPrefix + 1
else:
cntextra = cntextra + 1
print "A3 %d:%d %d:%c" % (len(k) , len(v) , ord(k[0]) , ord(k[0]))
print k
print k.encode('hex')
#exit(0)
print "Version %d" % cntver
print "Last Block %d" % cntlastblk
print "Last Fast %d" % cntlastfast
print "Last Header %d" % cntlastheader
print "<h><number><n> %d" % cnthnumbern
print "<h><number><hash> %d" % cnthnumberhash
print "<h><number><hash><t> %d" % cnthnumberhasht
print "<b><number><hash> %d" % cntbnumberhash
print "<H><hash> %d" % cntHhash
print "cnt32 %d" % cnt32
print "<hash>1 %d" % cnthash1
print "<r><number><hash> %d" % cntrnumberhash
print "<receipts-><hash> %d" % cntreceiptshash
print "<ethereum-config><hash> %d" % cntethereum
```



```

print "<setting-mipmap-version> %d" % cntminimap
print "<secure-key-> %d" % cntsecurekey
print "<mipmap-log-bloom> %d" % cntlog
print "cntBloomBits key %d" % cntBloomBits
print "cntBloomBitsIndexPrefix key %d" % cntBloomBitsIndexPrefix
print "TBD %d" % cntextra

print "Total number of key value pairs %d" % cnt
final = cntver + cntlastblk + cntlastfast + cntlastheader + cnthnumber +
cnthnumberhash + cnthnumberhasht + cntbnumberhash + cntHhash + cnt32 +
cnthash1 + cntrnumberhash + cntreceiptshash + cntethereum + cntminimap +
cntsecurekey + cntlog + cntBloomBits + cntBloomBitsIndexPrefix
print "Our Count %d" % final
print cnt == final

```

Output

```

Version 1
Last Block 1
Last Fast 1
Last Header 1
<h><number><n> 2688679
<h><number><hash> 2689507
<h><number><hash><t> 2689507
<b><number><hash> 2689375
<H><hash> 2740851
cnt32 135505530
<hash>1 13063629
<r><number><hash> 2689632
<receipts-><hash> 13114973
<ethereum-config><hash> 1
<setting-mipmap-version> 1
<secure-key-> 19547490
<mipmap-log-bloom> 2779
TBD 0
Total number of key value pairs 197421958
Our Count 197421958
True

```

This full-fledged program looks at every key-value pair in the 196 million pairs present in the leveldb database. It accesses every key-pair and increments a certain variable by 1 when a key is found. For example, we check if the key starts with an H and its length is 33 bytes. The `elif` statements everywhere speeds up the code. The reason being that, there are lots of keys that start with a r or n, for example.

At the very end is an `else` statement that kicks in when a key is not handled.

There is one more error check. For each key, the value of the `final` variable is calculated. It is a sum of all the keys. When the `final` variable and the variable `cnt` do not match, it signifies that a certain key is missed out. We ran this program in the beginning of the year 2017. The last two `if` statements check for Bloom keys which have been newly added.

```
Version 1
Last Block 1
Last Fast 1
Last Header 1
<h><number><n> 4437560
<h><number><hash> 4438135
<h><number><hash><t> 4438135
<b><number><hash> 4438135
<H><hash> 4438135
cnt32 39564377
<hash>1 73218831
<r><number><hash> 4438135
<receipts-><hash> 0
<ethereum-config><hash> 1
<setting-mipmap-version> 0
<secure-key-> 375168
<mipmap-log-bloom> 0
cntBloomBits key 2217984
cntBloomBitsIndexPrefix key 1084
TBD 0
Total number of key value pairs 142005684
Our Count          142005684
True
```

This output represents Byzantium at its very best. We have lots and lots of Bloom keys. But, we have 0 key starting with receipts. All code that depends upon receipts will stop working.

```
Version 1
Last Block 1
Last Fast 1
Last Header 1
<h><number><n> 4186894
<h><number><hash> 4186894
<h><number><hash><t> 4186894
<b><number><hash> 4186894
<H><hash> 4186894
cnt32 592754705
<hash>1 48643082
<r><number><hash> 4186894
<receipts-><hash> 0
<ethereum-config><hash> 1
<setting-mipmap-version> 1
<secure-key-> 34511842
<mipmap-log-bloom> 2807
cntBloomBits key 2091008
cntBloomBitsIndexPrefix key 1022
TBD 0
```

Lightning strikes the same place multiple times. The above output shows us over 700 million keys. All that we did was have an older Ethereum blockchain. We then ran `geth` and the `chaindata` folder crossed 260 GB in size. We know that this is Byzantium's output as the receipts key is 0. For the record, the Byzantium blockchain size is less than 40GB. The aim of this chapter was to talk about the various key-value pairs stored in the Ethereum leveldb database stored in the `chaindata` folder.

Let's write some quick and dirty code in Python to understand rip. The standard Python library `rip` brings in the code required for encoding numbers, strings and lists.

```
$sudo pip install rlp
```

```
ch3319.py
import rlp

a = rlp.encode(3)
print "%d length %d" % (ord(a[0]), len(a))
a = rlp.encode(127)
print "%d length %d" % (ord(a[0]), len(a))
a = rlp.encode(128)
print "%d:%d length %d" % (ord(a[0]), ord(a[1]), len(a))
a = rlp.encode(129)
print "%d:%d length %d" % (ord(a[0]), ord(a[1]), len(a))
a = rlp.encode(258)
print "%d:%d:%d length %d" % (ord(a[0]), ord(a[1]), ord(a[2]), len(a))
a = rlp.encode(65536 + 512 + 1 )
print "%d:%d:%d:%d length %d" % (ord(a[0]), ord(a[1]), ord(a[2]), ord(a[3]),
len(a))
print "_"
a = rlp.encode('AB')
print "%d:%c:%c length %d" % (ord(a[0]), ord(a[1]), ord(a[2]), len(a) )
a = rlp.encode('ABC')
print "%d:%c:%c length %d" % (ord(a[0]), ord(a[1]), ord(a[2]), len(a) )
a = rlp.encode('ABCDThis string is exactly 56 characters long. It is ....')
print "%d:%c:%c:%c Length %d" % (ord(a[0]), ord(a[1]), ord(a[2]), ord(a[3]) ,
len(a))
a = rlp.encode('ABCDThis string is exactly 57 characters long. It is ....')
print "%d:%d:%c:%c:%c Length %d" % (ord(a[0]), ord(a[1]), ord(a[2]), ord(a[3]) ,
ord(a[4]), len(a))
a =
rlp.encode('ABABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCD')
```

[illegible]

Output

3 length 1
127 length 1
129:128 length 2
129:129 length 2
130:1:2 length 3
131:1:2:1 length 4
—
130:A:B length 3
131:A:B length 4
184:8:A:B Length 58
184:57:A:B:C Length 59
184:236:65:B:A:B Length 238
—
195:130:A:B Length 4
196:131:A:B:C Length 5
198:130:A:B 130:C:D Length 7

Small numbers up to a value of 127 are not encoded, so they are left unchanged. The numbers 3 and 127 included are stored in their original form. The length of the encoded string is 1. The encode function from the library rlp is used extensively. All comments for code in this library show the numbers in hex, the range is shown as 0 to 0x7f.

The rules of rlp kick in when a number is greater than 127, let's say 128 is encoded. The length of the final string stored in variable `a` is now 2. The first byte is 129 which is to be read as 128 + 1. The 1 is the number of bytes following, i.e. 1 byte. This byte is the actual value of the number. The length of the encoded string is a total of 2 bytes.

When we encode a 129, the same rules apply a $128 + 1$ for length and the value follows 129.

For a large number like 258, 2 bytes are required to store its value, 2 and 1 if it's Little Endian and 1 and 2 for Big Endian. The length of the string is now 3 bytes, 130 is $128 + 2$ bytes for storing the actual values 1 and 2. This confirms that rlp uses Big Endian to store numbers. A number like $65536 + 512 + 1$ needs 3 bytes to store its value. The initial byte is $128 + 3$ bytes for the length or 131. The next three bytes are multiplied by $65536 + 256 * 2 + 1$.

Let's now consider simple strings. The string AB is less than 55 bytes long. So, 2 bytes needed to store the A and the B and $128 + 2$ gives us 130 as the first byte, followed by the actual string bytes.

The string ABC is one larger and thus the first byte is 131, $128+3$. Then comes a string that is on the borderline 56 bytes large. The first byte is $128 + 56 = 184$ and the rest of the string bytes follow. The first byte is now 57, the length of the string following and then the actual string. Since it is a slightly larger string, the first byte remains the same, 184. The next byte is now 236 as this is the length of the string.

For a list, the rules are different. For instance, there is a list of one element, AB. This is a small list and the list is stored as a string. When the size of the string is increased to ABC, then the inside string length changes from 130 to 131 and the outside list length also changes from 195 to 196. For a list of 2 elements, the length is now 7 and then each element is stored back to back.

The above explanation gives a fair understanding of rlp. The standard go libraries of Ethereum perform these tasks, so why worry.

CHAPTER 34

Ethereum Unravelling the State Machine

This chapter is again like all the other chapters, a very difficult one. Understanding the state machine in Ethereum is not just difficult, but difficulty raised to infinity. The only solace we found was in the source code but in a programming language we are not familiar with, go lang.

We want to know the account balance of a valid Ethereum address on the mainnet:

2c80814d41030380606fd8e2fb988dedd6aae1d9?

How is this account balance stored in the leveldb or in the Ethereum state machine? The intention is not to change the balance in this Ethereum address. Though, it is next to impossible to steal the Ether as no one has access to the private keys.

The transaction containing this ethereum address with a value of 1.13 ether is in block number 2505660. The blockchain explorer states that the block was mined on 25th October 2016. This transaction was created by ShapeShift, we gave bitcoins to them and we got ether in return. Any block number larger than 2505660 will show the same result.

We use block number 2688678 which was mined on the 25th of November 2016, and ask you to do the same. That's because we stopped downloading the Ethereum blocks on this date. The Ethereum source code looks at the LastBlock leveldb key to find the balance of an account.

Let's as always build on unravelling the heart of Ethereum's state machine. As a UTXO makes Bitcoin what it is, this state machine gives life to Ethereum.

A fair-weather warning. Again, it may not work for you. Our copy of Ethereum folder is over 80GB in size and the files have been tweaked at times. Our Ethereum folder is dated 25th November 2016. Anyways, you can learn from the concepts at least. Once again, even we have had a problem replicating what we did. Though has been warned.

Converting a Hash into a Block Number and Obtaining the All-Important State Root Member

```
ch3401.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/core/types"
)
func main() {
```

```

db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
hash, _ := db.Get([]byte("LastBlock"), nil)
fmt.Printf("Block Hash %x\n", hash)
key := append([]byte("H"), hash...)
data, _ := db.Get(key, nil)
blockno := binary.BigEndian.Uint64(data)
number := make([]byte, 8)
binary.BigEndian.PutUint64(number, uint64(blockno))
key1 := append([]byte("h"), number...)
key1 = append(key1, []byte("n")...)
hash1, _ := db.Get(key1, nil)
fmt.Printf("Block Hash %x\n", hash1)
key2 := append(append([]byte("h"), number...), hash1...)
fmt.Printf("Block Number %d\n", blockno)
data2, _ := db.Get(key2, nil)
header := new(types.Header)
tmp := bytes.NewReader(data2)
rlp.Decode(tmp, header)
fmt.Printf("%x\n", header.Root)
}

```

Output

```

Block Hash
8785106269eada467d1d94cb8e13b067609752a32f87aca0b87eeb05ae9391cb
Block Hash
8785106269eada467d1d94cb8e13b067609752a32f87aca0b87eeb05ae9391cb
Block Number 2688678
a8ecb15a79ea5d9b26220c2f0f4bb6b064fb913c43880b021ec780e3f0e42189

```

As before, the keywords package, import and func are ignored. To simply prove a point, first, the value of the key LastBlock is acquired. The return value is a block hash starting with 8785. As of now, we are clueless on what this block number and this hash represents.

Now, an H is added to this hash. This gives the block number. The block number returned is 2688678 and it is larger than the block, which contains the transaction that transferred ether to our Ethereum address, i.e. block number 2505660.

The task is to obtain the block hash of block number 2688678, even though we have it. As seen in the earlier program, the block number is converted into BigEndian, h is added, then comes the block number and finally end it with an n.

The return value of this key is a block hash which is stored in variable hash1. The hash1 value matches the hash of the LastBlock. Adding an h to this block hash gives the block header. The rlp is decoded and all the important state root or root member are displayed.

For some reason, the blockchain explorer does not find the State Root value important enough so it does not display it. The only way out is to use the eth.getBlock function. The last time we executed this program, we got a block number of 4443777.

Hardcoding the Block Number 2688678 in the Code to get to the state root

```
ch3402.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/core/types"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    blockno := 2688678
    number := make([]byte, 8)
    binary.BigEndian.PutUint64(number, uint64(blockno))
    key1 := append([]byte("h"), number...)
    key1 = append(key1, []byte("n")...)
    hash1, _ := db.Get(key1, nil)
    key2 := append(append([]byte("h"), number...), hash1...)
    data2, _ := db.Get(key2, nil)
    header := new(types.Header)
    tmp := bytes.NewReader(data2)
    rlp.Decode(tmp, header)
    fmt.Printf("%x\n", header.Root)
}
```

Output

a8ecb15a79ea5d9b26220c2f0f4bb6b064fb913c43880b021ec780e3f0e42189

Since understanding the state root is so very difficult, we have hard coded the block number in the variable blockno. So, if you are in sync with us, you will see the same output. Even if you are not, you will see the same output provided the block number is higher. When we ran the program much later, our LastBlock value was a tad higher at 4443777. Byzantium terrority.

All our codes hence forward will begin with the above lines, where we have the same value for the state root variable.

As a gentle reminder, any block number larger than 2505660 will give the same result, but the intermediate output will differ. A program like this will never give an error unless you have our magic number in your leveldb folder.

Understanding the Structure of State Root

```
ch3403.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
```



```

    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/core/types"
)
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
    blockno := 2688678
    number := make([]byte, 8)
    binary.BigEndian.PutUint64(number, uint64(blockno))
    key1 := append([]byte("h"), number...)
    key1 = append(key1, []byte("n")...)
    hash1, _ := db.Get(key1, nil)
    key2 := append(append([]byte("h"), number...), hash1...)
    data2, _ := db.Get(key2, nil)
    header := new(types.Header)
    tmp := bytes.NewReader(data2)
    rlp.Decode(tmp, header)
    root := header.Root[:]
    fmt.Printf("%T:%T\n", root, header.Root)
    data, abc := db.Get(root, nil)
    fmt.Printf("Size of Data %d:%s\n", len(data), abc)
}

```

Output

```

[]uint8:common.Hash
Size of Data 532

```

The member `header.Root` has a type of `common.Hash`. The datatype of variable `root` is an array of bytes or `uint8`. Golang is very picky about data types given to a function.

The type `Hash` is defined in the package `common` in the physical file `types.go`. Given below is a sampler.

```

package common

const (
    HashLength = 32
    AddressLength = 20
)

type (
    Hash [HashLength]byte
    Address [AddressLength]byte
)

```

The `const` keyword ensures that the value assigned to a variable does not change. Try and change it and go lang will scream murder. The variable `HashLength` is declared a `const` variable. A type `Hash` is created and it is an array of 32 bytes. The variable `root` has the same type as `header.Root` but its type is changed from `common.Hash` to `[]uint8`.

```

func (h Hash) Bytes() []byte { return h[:] }

```

The above statement displays how a Hash is converted to an array of Bytes.

The last Get function can be rewritten as

```
data , _ := db.Get(header.Root[:], nil)
```

All that we have learned so far, is that the state root is a 32-byte hash having an actual key in the leveldb database and it carries a value of 532 bytes. Let's decipher these bytes.

Unfortunately, if you are running Byzantium please read further but you will get no sensible output. That's because the size of the data returned is 0 bytes. The error message says key not found. This chapter as we said earlier, makes lots of assumptions, failure goes with the territory.

Someday we will figure out how to get the state data. If you are however working with a blockchain which you are upgrading to Byzantium, things will work. A fresh blockchain downloaded after the 14th of October has none of these leveldb keys.

The Value Portion of the State Root

```
ch3404.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/core/types"
)
var Children [16][[]byte]
func main() {
    db , _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata" , nil)
    blockno := 2688678
    //blockno := 2686699
    number := make([]byte , 8)
    binary.BigEndian.PutUint64(number , uint64(blockno))
    key1 := append([]byte("h") , number...)
    key1 = append(key1 , []byte("n")...)
    hash1 , _ := db.Get(key1 , nil)
    key2 := append(append([]byte("h") , number...) , hash1...)
    data2 , _ := db.Get(key2 , nil)
    header := new(types.Header)
    tmp := bytes.NewReader(data2)
    rlp.Decode(tmp, header)
    data , _ := db.Get(header.Root[:], nil)
    fmt.Printf("data length %d\n" , len(data))
    fmt.Printf("%x:%x\n" , data[0:38] , data[531:532])
    elems, temp , _ := rlp.SplitList(data)
    fmt.Printf("elemns length %d:%T temp %d\n" , len(elems) , elems, len(temp))
    fmt.Printf("%x:%x\n" , elems[0:1] , elems[1:33])
}
```

```

fmt.Printf("%x:%x\n", elems[33:34], elems[34:67])
fmt.Printf("%x:%x:%x\n\n", elems[525:526], elems[526:527], elems[527:528] )
for i := 0; i < 16; i++ {
    _, val, rest, _ := rlp.Split(elems)
    fmt.Printf("(%d) val %d rest %d %x:%x\n", i, len(val), len(rest), rest[0:1], rest[1:2])
    Children[i] = val
    elems = rest
    fmt.Printf("%x\n", Children[i] )
}
fmt.Printf("\n%x\n", Children[9] )
fmt.Printf("Elems %x:%d\n", elems, len(elems))
}

```

Output

```

data length 532
f90211a0c01f763a45be57967ea9962f988bde6efa4e51d51e277866ecc30e6f96f2deb4a
085:80
elemns length 529:[]uint8 temp 0
a0:c01f763a45be57967ea9962f988bde6efa4e51d51e277866ecc30e6f96f2deb4
a0:85ecc6445cac4c468c2a501ea1278474e6725c341011367fe0272580faced98a0
cc:e9:07

(0) val 32 rest 496 a0:85
c01f763a45be57967ea9962f988bde6efa4e51d51e277866ecc30e6f96f2deb4
(1) val 32 rest 463 a0:1e
85ecc6445cac4c468c2a501ea1278474e6725c341011367fe0272580faced98
(2) val 32 rest 430 a0:35
1e92576fab1506b6406f7ffea7126bf08c2c0835edbbbc0e48c5e4703b76e7a8
(3) val 32 rest 397 a0:58
35317870a5e27fe76332cf8564814076ad307d661736123820ef21ca5c2cab36
(4) val 32 rest 364 a0:ef
58d77d8170ee5c425b2a10c7995efe228180ab3c53d2c8ced236de7866ef7e04
(5) val 32 rest 331 a0:01
ef1d919447891fe12329a4307c84c377fcd38e34338ba700ba9b9479ddecbcc5
(6) val 32 rest 298 a0:7a
0108f82a2186e731b86f41695c8612ed7286afa2343dc354b34fee0170e5125d
(7) val 32 rest 265 a0:18
7a391461e5b86ad03d164d0c11cac5905110b02657634719f59e423edfbf7fdf
(8) val 32 rest 232 a0:b6
18ce94cc4630c06d7a5be676443b4a6d4f130ca652d8ad7f0f8c53929332e7bd
(9) val 32 rest 199 a0:46
b61f4044634f281bc499ae748e68a23a665fd3c5d3c2ba0bafda622f9e6685ea
(10) val 32 rest 166 a0:12
467d0be2b170a82ea13afe879b831298ff7ee0baf923d303f50663e119793ccd
(11) val 32 rest 133 a0:66
12b03fbd428cbab00b1ec8610fa770ed1896ff4580283d30ecee3157be3483af
(12) val 32 rest 100 a0:28

```

```
66cf06aa4cde7563d4281a21797e55655a065ff17f4510e5324425a1c8420f2d
(13) val 32 rest 67 a0:0f
28037a5a61ec223172a722981a169dd620383a5fd02ae5ba41abf1831743d6df
(14) val 32 rest 34 a0:81
0faa4f66e501b22c09c9eac6d23fbdd23e6f9649edd004960d78a5585a48f9a1
(15) val 32 rest 1 80:00
8136a95a3c805bcbddf6aacf83e5432b6b7ba58945aaea4b213c8c7978cce907
b61f4044634f281bc499ae748e68a23a665fd3c5d3c2ba0bafda622f9e6685ea
Elms 80:1
```

A Primer on Rlp Again.

The first byte of the data variable received is 0xf9 or 249. This first byte reveals that there is a list following the first byte and then is its length. The number 247 is subtracted from 249 resulting in a value of 2. The next two bytes have the length of the actual list, in Big Endian format.

The number 02 is multiplied by 256, so the answer is 512. The hex value of 0x11 is converted to decimal which is 17. Adding 512 and 17 gives a value of 529, the size of the list. The length of the value field is calculated by adding 3 bytes to 529 which results in 532, the size of the data we have.

The SplitList function in rlp performs two tasks. It first figures out the size of the list, 529 bytes, as we did. It then extracts these 529 bytes after dropping the first three header bytes and stores them in a variable. This list is stored in the elems variable. Since there is no data left, the temp variable is 0.

In other words, the function SplitList returns a list, minus the header bytes and any other data structures following.

The hash value stored in the list will always be 32 bytes large. This value may be a hash but to rlp, it is only a 32-byte string. As the length of the hash is always below the magic number 55, 128 is added to 32 resulting in a value of 160 or 0xa0. Thus, each hash begins with a 0xa0. There are 16 hashes in our list and each hash takes up 33 bytes. On multiplying 33 by 16, the answer is 528. We will explain this one extra byte 0x80 later.

To prove our point, the elems array has the first 3 bytes removed. It starts with a 0xa0 and then a series of 32 bytes. Then again, a 0xa0 and a series of 32 bytes. We have shown the hard-coded array offsets.

A for loop is brought in. The loop variable i begins with 0 and then increases by 1 until it reaches a value of 16. This for looks very different when compared to other programming languages.

The Split function is equally simple to write. It reads just the first byte 0xa0, it knows that the string is 32 bytes and it returns the first 32-byte string minus the 0xa0. You can compare the first two hashes shown in the for loop with the first two hashes displayed earlier.

The Split function has read 33 bytes, so it subtracts from the original length 529 - 33 (32 + 1) and returns a number 496, which is the size of the rest of the data minus the first string. For the last time, the first byte of the rest array starts with an a0, followed by the first byte of the hash.

The Split function dishes out individually, the 16 decoded hashes present in the value of the State Root key. Simultaneously, 16 arrays of bytes called Children are created to hold these hashes.

The next requirement is the second hash value. For this purpose, the elems array is replaced with the rest array as the first hash values in rest array are significant. Each read hash is also saved in the Children's array at a different offset. Accordingly, each hash member of the list is wheedled out.

The for loop ends and the size of the rest array has a 0x80 at the start. This is the same value that we promised to explain later. It is the last byte of the data array. This explains the extra 1 byte not accounted for.

The 9th member of the Children's array is displayed for good luck.

The Last Program With Code Placed in a Function

```

ch3405.go
package main
import (
    leveldb "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/core/types"
    //"github.com/ethereum/go-ethereum/common"
)
func getChild( index uint8 , root []byte ) []byte {
var Children [17][]byte
db , _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata" , nil)
data , _ := db.Get(root[:], nil)
elems, _ , _ := rlp.SplitList(data)
c , _ := rlp.CountValues(elems)
fmt.Printf("Index %d Count %d\n" , index , c)
for i := 0; i < 16; i++ {
    _ , val, rest, _ := rlp.Split(elems)
    Children[i]= val
    elems = rest
    //fmt.Printf("%x\n" , Children[i] )
}
db.Close()
return Children[index]
}
func main() {
    db , _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata" , nil)
    //blockno := 2688678
    blockno := 2686699
    number := make([]byte , 8)
    binary.BigEndian.PutUint64(number , uint64(blockno))
    key1 := append([]byte("h") , number...)
    key1 = append(key1 , []byte("n")...)
    hash1 , _ := db.Get(key1 , nil)
    key2 := append(append([]byte("h") , number...) , hash1...)
    data2 , _ := db.Get(key2 , nil)
    db.Close()
    header := new(types.Header)
    tmp := bytes.NewReader(data2)
    rlp.Decode(tmp, header)
    hashr := getChild(9 , header.Root[:])

```

```
    fmt.Printf("%x\n", hashr )
    hashr = getChild(9 , hashr)
    fmt.Printf("%x\n", hashr )
    hashr = getChild(0x0e , hashr)
    fmt.Printf("%x\n", hashr )
    hashr = getChild(0x0a , hashr)
    fmt.Printf("%x\n", hashr )
    hashr = getChild(0x0b , hashr)
    fmt.Printf("%x\n", hashr )
    hashr = getChild(0x0c , hashr)
    fmt.Printf("%x\n", hashr )
    hashr = getChild(0x0e , hashr)
    fmt.Printf("%x\n", hashr )
    hashr = getChild(3 , hashr)
    fmt.Printf("%x\n", hashr )
}
```

Output

```
Index 9 Count 17
25858022a26bdd2a757e2ef0e4b8b52b56c6a1d74607feb36db112ca44b679ad
Index 9 Count 17
cd07db68733035649cd6191d32e5ecbe03567119bbd527131febebee61f9ba33
Index 14 Count 17
d4e3d28202d6ede962f3ded371cf1e6f817187628d324ff4394d47970df367bc
Index 10 Count 17
2e7c8277ad5065893aef52e76b9098933d89267f86f8361350f86a20bf50788c
Index 11 Count 17
07e912e0b5503daeb21e4a1486d517b94c6b90225010417b583d45dde49222b9
Index 12 Count 17
472e2f52b612b561ed28f3064e4a0d9c19867157dedd7e701d2967ad0300e91a
Index 14 Count 17
52a2d413f30b889561ad6397b0f0c8d5e4eb64c1a2462e267a9e3463118b570f
Index 3 Count 2
```

This program has the same code of the earlier program. Please bear with us, we have lost some of our marbles. The leveldb folder is opened and after reading the state root, the leveldb database is closed. Golang panics if the database is not closed as leveldb is a single user database.

The function getChild in the above program performs the same task and has all the code of the last program. It takes a number and a hash value. This number is an index into the array of 16 children. The return value is the member at that location. The getChild function is called 7 times, each time with a different index. The actual index number is not important for now.

The function CountValues discloses the number of encoded values in the list. The return value in Count is 17 for all, except the last function, getChild when the index value is 3. When it is time to quit, Count has a value of 2.

Finding how an account balance is stored in the big state machine

```

ch3406.go
package main
import (
    leveldb "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/common"
    "math/big"
)

func getChild( index uint8 , root []byte ) []byte {
    var Children [17][]byte
    db , _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata" , nil)
    data , _ := db.Get(root[:], nil)
    elems, _ , _ := rlp.SplitList(data)
    for i := 0; i < 16; i++ {
        _ , val, rest, _ := rlp.Split(elems)
        Children[i]= val
        elems = rest
        //fmt.Printf("%x\n" , Children[i] )
    }
    db.Close()
    return Children[index]
}

type Account struct {
    Nonce uint64
    Balance *big.Int
    Root common.Hash // merkle root of the storage trie
    CodeHash []byte
}

func main() {
    db , _ := leveldb.OpenFile("/Users/vijaymukhi/Desktop/chaindata" , nil)
    //blockno := 2688678
    blockno := 2686699
    number := make([]byte , 8)
    binary.BigEndian.PutUint64(number , uint64(blockno))
    key1 := append([]byte("h") , number...)
    key1 = append(key1 , []byte("n")...)
    hash1 , _ := db.Get(key1 , nil)
    key2 := append(append([]byte("h") , number...) , hash1...)
    data2, _ := db.Get(key2, nil)
    db.Close()
    header := new(types.Header)
    tmp := bytes.NewReader(data2)

```

```
rlp.Decode(tmp, header)
hashr := getChild(9, header.Root[:])
hashr = getChild(9, hashr)
hashr = getChild(0x0e, hashr)
hashr = getChild(0x0a, hashr)
hashr = getChild(0x0b, hashr)
hashr = getChild(0x0c, hashr)
hashr = getChild(0x0e, hashr)
db, _ = leveledb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
buf, _ := db.Get(hashr, nil)
fmt.Printf("Length of buf %d %x\n", len(buf), buf[0:20])
elems, _, _ := rlp.SplitList(buf)
fmt.Printf("Length of elems %d %x\n", len(elems), elems[0:20])
val0, rest, _ := rlp.SplitString(elems)
fmt.Printf("Length of val0 %d %x\n", len(val0), val0[0:30])
fmt.Printf("Length of rest %d %x\n", len(rest), rest[0:20])
val, _, _ := rlp.SplitString(rest)
fmt.Printf("Length of val %d %x\n", len(val), val[0:30])
var data Account
rlp.DecodeBytes(val, &data)
fmt.Printf("%d\n", data.Balance)
i := big.NewInt(1138357020000000000)
if i.Cmp(data.Balance) == 0 {
    fmt.Printf("Balance Matches\n")
}
fmt.Printf("Nonce %d\n", data.Nonce)
fmt.Printf("Root %x\n", data.Root)
fmt.Printf("Code Hash %x\n", data.CodeHash)
}
```

Output

```
Length of buf 112 f86e9d33f490493d5983ba51e6b6aa93c4e03388
Length of elems 110 9d33f490493d5983ba51e6b6aa93c4e03388461a
Length of val0 29
33f490493d5983ba51e6b6aa93c4e03388461a1145b0d32fe005e7012eb8
Length of rest 80 b84ef84c80880fcc41adacac1800a056e81f171b
Length of val 78 f
84c80880fcc41adacac1800a056e81f171bcc55a6ff8345e692c0f86e5b
11383570200000000000
Balance Matches
Nonce 0
Root 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cad001622fb5e363b421
Code Hash c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
```

Finally, everything works as advertised. Our account that we displayed earlier has the same account balance shown in this output.

How did we get here? From the last example we learnt that the last hash had a count of 2 and not 17.

The length of the data returned in the variable `buf` is 112 bytes. The first byte is `f8` and the second byte is `6e` or 110 bytes following. Therefore, this data has the size of 112 bytes, $110 + 2$.

The `SplitList` function knocks off the first 2 bytes, so there are 110 bytes left now. The source code of the file `raw.go` has code of `SplitString`.

The first condition in the original source code is that the first byte must be less than `0x80`. This test fails as our first byte is `0x9d`.

The next case statement checks for the byte to be less than `0xb8`. This test passes. The value of `0x9d` is subtracted from `0x80` to give a value of 29. Thus, the length of the `val0` variable is 29. Here again, the first byte is ignored and the next 29 bytes are read. We know all this because we read the Ethereum go source code. You can ignore this explanation as we assume that you have no access to the source code.

The rest variable now has 80 bytes. The first byte is `b8`, thus it fails the second case statement as they are not equal. The third check is for the value to be less than `0xc0`. This test succeeds.

Now to subtract `0xb8` from `0xb7`. This resultant value of 1 is added to 1, thus resulting in a value of 2. This is the value of the variable `tagsize` (from the source). The `contentsize` (source) variable is calculated by converting the first byte of the string which is a number `0x4e` or 78, as the content size. The second byte in the line 'Length of rest'.

The `tagsize` variable is used as the starting point and the `contentsize` variable as the ending point to avail the second string.

Back to earth. Then, an object called `data` that looks like an `Account` type is created. This type or structure called `account` has four members. The first crucial member is the `Balance`. A `NewInt` object is created from the `math/big` package and then the `Cmp` method is used to compare the two values. We get a match.

The `Nonce` field displayed on the screen shows a value of 0, thus indicating that this address has not been used till now. The `Root` member has been displayed earlier, what's pending is the `CodeHash` field.

Our code is not portable and it is complex, we tried our best to simplify it.

This will not work with any Ethereum address but the one we hard coded. In a generic program, we cannot assume that the `getChild` function will get called 7 times with these index values.

Finding the Sha Hash Value of the Ethereum Address

```
ch3407.go
package main
import (
    "hash"
    "bytes"
    "fmt"
    "sync"
    "github.com/ethereum/go-ethereum/crypto/sha3"
    "github.com/ethereum/go-ethereum/common"
)
type hasher struct {
    tmp *bytes.Buffer
    sha hash.Hash
}
```

```
func newHasher() *hasher {
    h := hasherPool.Get().(*hasher)
    return h
}
var hasherPool = sync.Pool{
    New: func() interface{} {
        return &hasher{tmp: new(bytes.Buffer), sha: sha3.NewKeccak256()}
    },
}
func compactHexDecode(str []byte) []byte {
    l := len(str)*2 + 1
    var nibbles = make([]byte, l)
    for i, b := range str {
        nibbles[i*2] = b / 16
        nibbles[i*2+1] = b % 16
    }
    nibbles[l-1] = 16
    return nibbles
}
func main() {
    var hashKeyBuf [33]byte
    var index [64]byte
    key := common.Hex2Bytes("2c80814d41030380606fd8e2fb988dedd6aae1d9")
    h := newHasher()
    h.sha.Reset()
    h.sha.Write(key)
    buf := h.sha.Sum(hashKeyBuf[:0])
    fmt.Printf("Len %d\n", len(hashKeyBuf[:0]))
    fmt.Printf("%x\n", buf)
    buf1 := compactHexDecode(buf)
    fmt.Printf("%x\n", buf1)
    for i := 0 ; i < 64 ; i++ {
        index[i] = buf1[i]
    }
    for i := 0 ; i < 10 ; i++ {
        fmt.Printf("%02x ", index[i])
    }
}
```

Output

Len 0

99eabce3f490493d5983ba51e6b6aa93c4e03388461a1145b0d32fe005e7012e
09090e0a0b0c0e030f0409000409030d050908030b0a05010e060b060a0a09030c040e0
0030308080406010a010104050b000d03020f0e0000050e070001020e10
09 09 0e 0a 0b 0c 0e 03 0f 04

This program peels away one more layer of the onion. It's time to explain the 7 magic numbers we ignored earlier.

First, the Ethereum address starting with 2c8081 is converted into a series of bytes. Then, a function called `newHasher` is executed. In the original code, this same function takes two parameters, both have a value of 0.

This complicated `golang` code simply returns an empty hasher type which has only two members. The second member is more important, it is a sha hash value.

Some time ago, there was chaos in the crypto world. Everyone suggested that the sha hash used in Bitcoin must be replaced with a new sha algorithm. One of the algorithms invented though not used by Bitcoin was called sha3 or Keccak. Ethereum decided to use sha3 and not sha2.

The three lines of code calculates the sha3 hash given our Ethereum address. We will not go into major details here. The `Reset` function resets everything and the `Write` function calculates the sha hash value of the Ethereum address passed to it. The `Sum` function is passed a variable of size 0, not relevant for now.

Ultimately, the `buf` variable has the sha3 hash value stored in it. Now, adding our two bits. The function `compactHexDecode` is called to convert this hex string or hash into individual bytes.

The first byte of the sha hash is 99. An array of bytes 64 large is created and each member of this hash array is stored into a corresponding member of the index array. In short, an array called `index` is created to store each individual digit as a number. These numbers now look very familiar. This explains the numbers used when calculating the index for the `getChild` function. This example has too many hardcoded values.

The Account Balance With Some Error Checks

```
ch3408.go
package main
import (
    leveldb "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "encoding/binary"
    "github.com/ethereum/go-ethereum/rlp"
    "bytes"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/common"
    "math/big"
    "github.com/ethereum/go-ethereum/crypto/sha3"
    "sync"
    "hash"
)
type hasher struct {
    tmp *bytes.Buffer
    sha hash.Hash
}
func newHasher() *hasher {
    h := hasherPool.Get().(*hasher)
    return h
}
var hasherPool = sync.Pool{
    New: func() interface{} {
        return &hasher{tmp: new(bytes.Buffer), sha: sha3.NewKeccak256()}
```

```
    },
}

func compactHexDecode(str []byte) []byte {
    l := len(str)*2 + 1
    var nibbles = make([]byte, l)
    for i, b := range str {
        nibbles[i*2] = b / 16
        nibbles[i*2+1] = b % 16
    }
    nibbles[l-1] = 16
    return nibbles
}

func getChild( index uint8 , root []byte ) ([]byte , int) {
    var Children [17][]byte
    db , _ := leveledb.OpenFile("/Users/vijaymukhi/Desktop/chaindata" , nil)
    data , _ := db.Get(root[:], nil)
    elems, _ , _ := rlp.SplitList(data)
    c , _ := rlp.CountValues(elems)
    for i := 0; i < 16; i++ {
        _ , val, rest, _ := rlp.Split(elems)
        Children[i]= val
        elems = rest
        //fmt.Printf("%x\n" , Children[i] )
    }
    db.Close()
    return Children[index] , c
}

type Account struct {
    Nonce uint64
    Balance *big.Int
    Root common.Hash // merkle root of the storage trie
    CodeHash []byte
}

func main() {
    address := "2c80814d41030380606fd8e2fb988dedd6aae1d9"
    //address := "8c4e0c160a0d8c17dfba4b04e061b42767e4abe9"
    db , _ := leveledb.OpenFile("/Users/vijaymukhi/Desktop/chaindata" , nil)
    blockno := 2686699
    number := make([]byte , 8)
    binary.BigEndian.PutUint64(number , uint64(blockno))
    key1 := append([]byte("h") , number...)
    key1 = append(key1 , []byte("n")...)
    hash1 , _ := db.Get(key1 , nil)
    key2 := append(append([]byte("h"), number...), hash1...)
    data2, _ := db.Get(key2, nil)
    db.Close()
}
```

```

header := new(types.Header)
tmp := bytes.NewReader(data2)
rlp.Decode(tmp, header)

var hashKeyBuf [33]byte
var index [64]byte
key := common.Hex2Bytes(address)
h := newHasher()
h.sha.Reset()
h.sha.Write(key)
shahash := h.sha.Sum(hashKeyBuf[:0])
fmt.Printf("%x\n", shahash)
nibbles := compactHexDecode(shahash)
fmt.Printf("%x\n", nibbles)
for i := 0 ; i < 64 ; i++ {
    index[i] = nibbles[i]
}
hashr := header.Root[:]
var cnt int
for i := 0 ; i <= 63 ; i++ {
    prevhashr := hashr
    hashr, cnt = getChild(nibbles[i], hashr)
    fmt.Printf("hashr %x cnt %d\n", hashr, cnt)
    if cnt == 2 {
        hashr = prevhashr
        break
    }
}
db, _ := leveledb.OpenFile("/Users/vijaymukhi/Desktop/chaindata", nil)
buf, _ := db.Get(hashr, nil)
elems, _, _ := rlp.SplitList(buf)
_, rest, _ := rlp.SplitString(elems)
val, _, _ := rlp.SplitString(rest)
var data Account
rlp.DecodeBytes(val, &data)
fmt.Printf("%d\n", data.Balance)
i := big.NewInt(1138357020000000000)
if i.Cmp(data.Balance) == 0 {
    fmt.Printf("Balance Matches\n")
}
fmt.Printf("Nonce %d\n", data.Nonce)
fmt.Printf("Root %x\n", data.Root)
fmt.Printf("Code Hash %x\n", data.CodeHash)
}

```

Output

address := "2c80814d41030380606fd8e2fb988dedd6aae1d9"

```
99eabce3f490493d5983ba51e6b6aa93c4e03388461a1145b0d32fe005e7012e
09090e0a0b0c0e030f0409000409030d050908030b0a05010e060b060a0a09030c040e0
0030308080406010a010104050b000d03020f0e0000050e070001020e10
hashr f0d802457b44fffe5803007f19ddb93d1e5644175d397e1b62fea2b55f45817e
cnt 17
hashr a973a333b4b971e29c239d3d02a4e4b63d55defe3e29f774e0a455186751cdc4
cnt 17
hashr 5619cd63a56d1822b81b6ab0409b26087dda222c5be33b19481b13609e3070c9
cnt 17
hashr 0630936ba61acf8958f5aaadaf6bf172657fc6dfbc3019394f2c5c6b4eafc91
cnt 17
hashr 07e912e0b5503daeb21e4a1486d517b94c6b90225010417b583d45dde49222b9
cnt 17
hashr 472e2f52b612b561ed28f3064e4a0d9c19867157dedd7e701d2967ad0300e91a
cnt 17
hashr 52a2d413f30b889561ad6397b0f0c8d5e4eb64c1a2462e267a9e3463118b570f
cnt 17
hashr cnt 2
1138357020000000000
Balance Matches
Nonce 0
Root 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cad001622fb5e363b421
Code Hash c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
address := "8c4e0c160a0d8c17dfba4b04e061b42767e4abe9"
388c5aa0fc608a2bdb7891ded9ae1d5c204cefc64a704dfd20a6f29f26cab4e9
0308080c050a0a000f0c0600080a020b0d0b070809010d0e0d090a0e010d050c0200040
c0e0f0c06040a0700040d0f0d02000a060f02090f02060c0a0b040e0910
hashr 207655097d9a66c4eaf40debbac552ca65ba201efb26d684599714ff1285cd51
cnt 17
hashr 4a0b7d5318a94359494c384930ed977f8b8ab9e6bc89204748ccf6e8bf90228c
cnt 17
hashr d1696b754d2523758909d8a9d5e9b8ef91ffb8d7544d622fd38a0a42d2e40ff9
cnt 17
hashr e7db701f698b664beba88e4a7487e71e8a3349e779f39f45cfdbf3b510968daa
cnt 17
hashr 48baca28383d15f317f11688336ee239ed5379d8267ae85e73873709dadc16a1
cnt 17
hashr 0a7a67e37bc2029812186f5f71942d2f14dbc945697f8476c9e87ab79e4db6a4
cnt 17
hashr a96e9e46ac39a1ffeedbcce588e0b3adb10b8ec36e76e8e1a669050a74a8319f
cnt 17
hashr b2f10455e9aebd38e7173f05a4ef83a5d334bdee279d72fd634a348ebba7543f
cnt 17
hashr cnt 2
1115706210000000000
Nonce 0
```

```
Root 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cad001622fb5e363b421
Code Hash
c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
```

This program is a culmination of all the code written so far. To set things right, two Ethereum addresses with valid balances are used. These are :

```
2c80814d41030380606fd8e2fb988dedd6aae1d9,
8c4e0c160a0d8c17dfba4b04e061b42767e4abe9.
```

Our output shows both these addresses, and you can see the difference in the displayed hash values. If it works for two Ethereum addresses does not indicate that it will work for all.

The code from the last example, which calculates the sha3 hash of the Ethereum address is brought in. The array index, that now contains the individual offsets or indexes is restored into the 16-hash array list.

Then, in the for loop, the getChild function is called with 2 parameters, the index into the array which acts like a key to fetch the hash list and the hash. The return value is the hash value the index points to and the value in Count, which is a count of elements in the list. Normally, the value in Count is 17 in the beginning and towards the end, it is 2. We will not talk about a modified Patricia Merkle tree.

When the count shows 2, it is time to go. Since the previous value of the hash are crucial, it is saved in the prevhashr variable at the very beginning. When count is 2, prior to quitting, the hash variable is set to the value prevhashr.

This value of hashr is used to fetch, again, the value associated with the key hashr. The same task is repeated.

```
ch3409.go
package main
import (
    "github.com/syndtr/goleveldb/leveldb"
    "fmt"
    "github.com/ethereum/go-ethereum/rlp"
    "github.com/ethereum/go-ethereum/common"
    "math/big"
)
type Account struct {
    Nonce uint64
    Balance *big.Int
    Root common.Hash // merkle root of the storage trie
    CodeHash []byte
}
func main() {
    db, _ := leveldb.OpenFile("/Users/vijaymukhi/Library/Ethereum/geth/chaindata",
    nil)
    key :=
    common.Hex2Bytes("febc9da8352593c01e2ed404befcaa698a62789d12c705ea92688
    b3c5b30f5c")
    data, _ := db.Get(key, nil)
    data = data[35:]
    fmt.Printf("data length %d:%x\n", len(data), data)
```

```
    var acc Account
    rlp.DecodeBytes(data, &acc)
    fmt.Printf("Balance %v\n", acc.Balance )
}
>go run ch3409.go
data length
78:f84c02880fbf61e512358803a056e81f171bcc55a6ff8345e692c0f86e5b48e01b996ca
dc001622fb5e363b421a0c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bf
ad8045d85a470
Balance 1134733267601557507
```

The hash beginning with febc is used as a key and it returns 113 bytes of data. The source code tells us that for this node type the first 35 bytes must be removed. We do the same and then decode the bytes. We get an Account data type filled up with the balance and more.

We should have explained why things went wrong but we have no control over time.

Conclusion

We have had a hard time understanding the Ethereum state machine. Now we understand why the UTXO set and the state machine is what defines Bitcoin and Ethereum.

It's been quite a journey. We figured many things along the way. Like, the file blockchain.go is the most important file. The function loadlastState is called early in the day. There is a function statedb.GetBalance(addr). This function is called and it is given our address, present in the addr variable. The Exit function quits out from the package os. The Printf function is scattered everywhere to understand the golang code that creates the above magic.

Our apologies as we must end our topic here as we have a brain freeze. Feel free to make this code handle all cases, hoping it has worked so far.

CHAPTER 35

Bitcoin Cash vs Segwit vs Segwit2x

August 1st was a red-letter day for all of us. The high and mighty Bitcoin blockchain split into two. The original blockchain is yet known as Bitcoin or BTC and its smaller rival is called Bitcoin cash or BCC or BCH or XBC.

What's so unique about Bitcoin Cash? First and foremost, it's the block size, the 1MB limit is now 8MB. It allows 8 times more transactions to be stored in a block, thus enabling faster Bitcoin confirmations and obviously smaller fees. A block will be created in the same 10 minutes as before. But more transactions will fit in a Bitcoin Cash blockchain than in the Bitcoin Core blockchain.

From the website, www.bitcoinabc.org, we can download either the source code or the GUI of Bitcoin Cash. We downloaded the source, built it and then executed the bitcoind server, which downloaded the entire block chain, like before.

Tons of patience required here as you got to download 200 GB of data and it is not everyone's cup of tea.

We like to pat ourselves on our backs as we had saved an earlier backup of the Bitcoin block chain, dated 1 June 2017. So, the downloading didn't start from scratch but from 1st June onwards. We did not have too many blocks to download.

We first create a very simple table called size1 as shown below

```
create table size1 (blockno integer, size integer, time1 varchar);
```

```
ch3501.py
from cfuncs import *
import psycpg2
import time
conn = psycpg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
no = 0
for fno in range(0,60000):
    fname = "/Users/vijaymukhi/Library/Application Support/Bitcoin/blocks/blk%05d.dat" % fno
    try:
        f = open(fname, "rb")
        print "File %s" % fname
    except:
        break
    while ( True ):
        try:
            magic = rint(f)
            if magic == 0:
                continue
```

```
except:
break
size = rint(f)
header = f.read(80)
(ver,prevhash , merklehash , time1, diff, nonce) = struct.unpack("l32s32slll" , header[0:80])
blkhash = chash(header)
s1 = "Insert into size1 values (%d , %d , '%s' )" % (no , size , time.ctime(time1))
cur.execute(s1)
no = no + 1
f.seek(size -80 , 1)
conn.commit()
```

This program has been explained many times in the past. It reads the block header size and writes it to the size1 table along with the block date.

Let's run another SQL statement to return the 5 largest block sizes in the Bitcoin Cash Block Chain.

```
select blockno, size, time1 from size1 order by size DESC limit 5;
```

blockno	size	time1
479481	7998130	Wed Aug 16 17:24:13 2017
485802	7080904	Sun Sep 10 17:32:00 2017
479478	4830803	Wed Aug 16 21:12:11 2017
478571	4683334	Wed Aug 2 21:50:19 2017
484241	4229688	Thu Aug 31 02:16:14 2017

The output shows a block which is 7.99MB large and it is created on the 16th of August 2017. This proves that block sizes now can be as large as 8MB. Seeing is believing.

If you owned 5 Bitcoins on the 30th of July and slept throughout the day on the 31st of July, you will wake up all flabbergasted on the 1st of August as there will be two parallel Bitcoin chains. Your bitcoins will be present on both. Now you decide which chain your Bitcoins should be in. This is what happens when a new blockchain is not started from scratch.

From 1st of August 2017, they are two different currencies. If you decide to buy some Bitcoins on the Bitcoin Cash chain, these Bitcoins will not be available on the main Bitcoin network and obviously vice versa. You cannot spend your Bitcoins on both networks.

The header file has these lines in the original bitcoin core source code:

```
script/interpreter.h.
enum
{
    SIGHASH_ALL = 1,
    SIGHASH_NONE = 2,
    SIGHASH_SINGLE = 3,
    SIGHASH_ANYONECANPAY = 0x80,
};
```

These are the only four valid signature hash types on the Bitcoin blockchain. Let's look at the same enum in the Bitcoin Cash source code.

```
/** Signature hash types/flags */
enum
{
    SIGHASH_ALL = 1,
    SIGHASH_NONE = 2,
    SIGHASH_SINGLE = 3,
    SIGHASH_FORKID = 0x40,
    SIGHASH_ANYONECANPAY = 0x80,
};
```

The same enum now has 5 types, an extra type or flag SIGHASH_FORKID having a value 0x40 is newly added. Every Bitcoin Cash transaction carries this flag. So, these transactions will be rejected outright by the original bitcoin blockchain as it does not recognize this transaction type. For a transaction to be valid in BTC, it must have one of 4 hash types.

This way, Bitcoin Cash eliminated replay attacks. It only accepts a Bitcoin transaction that has this flag set. An original Bitcoin transaction created by a wallet will never have this flag set and hence it will not be accepted by Bitcoin Cash. On the other hand, a staunch Bitcoin Core miner will also reject a transaction type of 0x40.

The ABC means an Adjustable Blocksize Cap. Block number 478558 is the turning point, when Bitcoin Cash was born. This is block zero.

There is also no code for bringing in Segregated Witnesses. In other words, the divorce took place because of SegWit.

Now it's time to review the happenings in the Bitcoin universe. Not all of this makes for pretty reading, too much politics and we are bemused as we have no idea whom to believe.

But first let's start from the very beginning, way before Bitcoin was released. Most of you may/may not have heard about Cryddit, but he was one of the first few people who reviewed the Bitcoin source.

On a forum post:

<https://bitcointalk.org/index.php?topic=946236.msg10388435#msg10388435>

, this is what he said.

"I'm the guy who went over the blockchain stuff in Satoshi's first cut of the bitcoin code. Satoshi didn't have a 1MB limit in it. The limit was originally Hal Finney's idea. Both Satoshi and I objected that it wouldn't scale at 1MB. Hal was concerned about a potential DoS attack though, and after discussion, Satoshi agreed. The 1MB limit was there by the time Bitcoin launched. But all 3 of us agreed that 1MB had to be temporary because it would never scale."

After reading these lines, we wonder why the blockchain had a hard fork and why was there a world war over block sizes. Life will keep throwing curved balls at us, either way.

This was a hard fork or HF. As this fork was activated by a user, it is called a User Activated Hard Fork or UAHF. You will see this acronym all the time.

Let's first understand a hard fork vs a soft fork. But first hard forks.

We showed you earlier that Bitcoin Cash created a 7.99MB block which is against the rules of the old Bitcoin. The Bitcoin blockchain cannot have a block size larger than 1 MB. This means that what was invalid in the past is now valid under the new regime. We are changing or expanding the rules of the block chain. However, there is no backward compatibility with hard forks. Earlier version of the Bitcoin software will simply not run as they cannot handle a block size which is larger than 1 MB and 8 MB is definitely a no-no.

What is valid in Bitcoin Cash is not valid on the Bitcoin Core blockchain. Also, the Bitcoin Core miners will refuse such a block size as the cardinal rule of a 1MB block is broken. No miner will have the guts to mine blocks larger than 1 MB on the bitcoin core network.

A similar situation turned out between Ethereum and Ethereum classic. Lots of ether got stolen, Ethereum forked into 2. Anyone can do a hard fork, it's the miners support that decides the success or failure of a hard fork.

We created a Mukhi Coin that had a block size limit of 128MB. No miner touched us and Mukhi Coin died a slow and painful death. Okay, not true but you get the point.

That's why the jury is out on favoring Bitcoin Cash. Its 10% of the price of Bitcoin while writing this chapter but time alone will decide its fate. It's too early to take a call on Bitcoin Cash as in November 2017 we will see one more UAHF and then have a decision on the continuation of Bitcoin Cash.

There is a divide seen amongst the miners. Most miners do not support Bitcoin cash. Some miners have decided to exclusively mine Bitcoin Cash only, some decided to mine both. A miner that supports both Bitcoins will thus run two different pieces of software, one from Bitcoin Core the other from Bitcoin Cash. If miners make more money from mining Bitcoin Cash, more miners will choose Bitcoin Cash or over Bitcoin Core. Finally, the price of Bitcoin Cash versus Bitcoin Core will decide who wins. Nevertheless, the jury is still out.

There is plenty of politics of play in the hard fork of Bitcoin Cash and the main reason just flies of the window. If Satoshi were active today, none of this would have happened.

Let's start once again from scratch.

Nobody really knows and will ever know, what is going on in the Bitcoin blockchain. There is total chaos, no steady state at all. This is the beauty of a distributed consensus system. Miners keep passing transactions to other miners and nobody keeps any count of the transactions floating around.

Every miner is busy calculating a mindless hash. Suppose miner A finds a hash value that meets the rules, he broadcasts this fact to every miner. As there are miners all over the world, it takes time for this victory to filter through to all miners. There is also no one right answer, the winning hash can take many values. This is decided by the difficulty. All this has been explained earlier.

Now assume in another part of the world, another miner found a hash that is different from what miner A found. Both miners have found the winning hash almost at the same time. Now the fun starts, we have two competing chains. Some miners will mine on chain A, some on chain B, we have a fork. There are more than one chain many times in a blockchain's life cycle. After some time, one chain will be larger and therefore that would be the winning chain. The smaller chain dies and all transactions in this smaller chain are reconfirmed.

The Bitcoin blockchain knows how to handle such orphan blocks. This is in the very nature of Bitcoin. The Bitcoin blockchain may split 10 times a day and nobody will raise their eyebrow. If there are no forks then only will people wake up and smell the coffee.

Everything finally boils down to a single blockchain. And life continues as normal. The only people who loose in this game are the ones who bet on the wrong chain.

So, for a miner, always stay with the largest chain, stay away from invalid transactions in the block. Simply be a good person and play by the rules.

Now let's get to a Soft Fork.

A soft fork is a pain in the neck for a developer, a hard fork is the easy way out. People today still use Bitcoin Core Versions 11 and 12 and 13 and 14 at the same time. Each version of Bitcoin Core introduces some new features but the older versions gel in seamlessly. Though the newer features do not show up in the older versions, but they still work. A

soft fork is another name for backward compatibility where on software upgradation, the older guys yet work but not at full potential.

We are more interested in cases where a programmer makes a change to the source code and creates a User Assisted Soft Fork or a UASF.

If segregated witnesses were introduced in bitcoin core version 13.0, then Core versions 12.0 and 13.0 would see a different block chain. The segregated witness data will not be seen by version 12.0. but life will continue as before. These older versions will never witness a segregated version transaction. Newer features will only be seen by the newer versions.

We will understand soft forks and backward compatibility better when we get to the heart of segregated witness.

Let's take the same block size example again. We assume that the Bitcoin Core developers, in the newer version reduced the maximum block size to 500K from 1MB. We can expect some chaos with the older version as the question asked would be 'why are there no block larger than 500K when the upper limit is 1MB? Miners will gripe over wine but it's all in a hard day's work. Frankly speaking in such situations, they do not care about it, as in why is no one using the entire block. Conversely, if some miners using the older Bitcoin Core software (without the 500k size restriction) still produce new 1MB blocks then the newer Bitcoin core miners will reject them. The winner is the one with high horse power.

Let's assume that the miners who support the 500K blocks are in a minority. Their chain will be smaller as they reject blocks which are larger than 500K. Eventually, blocks in that chain will be rejected. As said earlier, stay with the majority. The result is two chains, which could result in a hard fork. Thus, both soft and hard forks produce a split in the blockchain. Money wins at the end of the day. The difference is that we see soft forks happening day in and day out. A hard fork happens only on a red-letter day.

If invalid transactions are found in a winning block, that block is marked invalid. So, a miner does not add invalid transactions in a block as he stands to lose his mining rewards for finding the winning hash.

A soft fork is preferred anytime as nobody wants a split. The history of Bitcoin is riddled with soft forks, SegWit being the latest and biggest.

Segregated Witnesses is the biggest change brought about in the bitcoin ecosystem. Segwit2x is Segregated Witness plus an increase in block size, it is likely to happen in November. SegWit was available on the Bitcoin core Testnet since the year 2016.

For various reasons the miners refused to endorse SegWit as it needed 95% of the mining power to be activated. Change in the Bitcoin world simply stopped. Then the bigger players in the space met at Coin Desk's Consensus 2017 conference in May. There they hammered out a solution called SegWit2x.

Segwit2x was deployed on the testnet network on July 14th, 2017. Live adoption started from 21st and on 1st August 2017, it was activated. One of the reasons why Bitcoin Cash was also launched on the very same day. We are in end October 2017 and the first half of Segwit2x has been activated as majority of miners have given a go.

Segwit2x is available for download using a Bitcoin client called BTC1. We will use this version later. The bitcoin client BTC1 is a fork of the Bitcoin Core source. SegWit remember, is a Bitcoin Core invention.

Is the circus over... no. The Bitcoin Core developers are happy that segregated witness is alive and happy and kicking. What they are upset about is the increase in block size. So, come November, they may not upgrade their software for the hard fork. It is not guaranteed that this November hard fork will ever get approval. But SegWit is activated.

The decision makes no difference as most of the future innovations of Bitcoin like Sidechains depend upon Segregated Witnesses and not on a 2MB block size increase. As it is, Segwit increases the block size but not as much as some miners want.

Finally, we may be left with three Bitcoin chains, the original Bitcoin Core, Bitcoin Cash and a Segwit2x chain. There are also hard forks like Bitcoin Gold that we have quietly ignored. There will be many more.

Back to basics. What is a Segregated Witness?

In the history of Bitcoins, there have been many incremental changes to the basic ideas. These ideas have stood the test of time. This is the first time a major overhaul has been attempted and that code is called Segregated Witness. It solves the most critical problem of transaction malleability where two different transaction hashes can belong to the same transaction. Such an eventuality takes place when the script code in the output performs the same task but uses different opcodes. The transaction does what it does, but its hash is different.

In our view, transaction malleability was overkill but people treated it as a major unresolved security issue. The major advantage of Segregated Witness is that it rewrote how transactions are structured. One of the emerging technologies that will allow Bitcoins to enter the world of micro payments and other payment networks and more is the concept of the Lightning network or better still Sidechains. Payments can then be made in seconds and not hours. It will change the way the world uses the blockchain. A lot of the implementations of the Lightning network hinges on the presence of Segregated Witness.

The Bitcoin Core wants Bitcoins to be a settlement layer, the others want Bitcoin to stay as a crypto-currency. Please take sides.

There is nothing wrong with the concepts or the code behind Segregated Witness. The issue is simple; politics. There is an ongoing war on deciding the size of a Bitcoin block. This is the 1MB controversy. At the same time, more than 95% of the miners must approve the code of Segregated Witness. Democracy at its worst. Miners must show their acceptance by downloading version 0.13 or higher and running this version.

Fortunately for us, big business is driving SegWit2x and not the Bitcoin Core. The flip side is that programmers lose control of the direction Bitcoin will take.

This is a very complex issue and we could write a book on which side is right. Both sides have a point of view, we lose.

Segregated Witness was available on the testnet for a very long time, please add the line `testnet=1` in the file `bitcoin.conf`. You could use mainnet like we did in the past, but in the last 3 years that we have been using Bitcoins, we cannot waste good money over nothing.

Then run the following command.

```
$bitcoin-cli getrawtransaction
6cf13cee271e49db5e97546b91c8c48bd0b928c137dfe4b734bc7e405c98ab7d 1
```

Output

```
{
  "hex":
    "01000000000101c372285682f939e6426566018bc9f97bbc8fd926b1fe5eaaaf6d76ea5c80e80
    a0000000023220020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58f
    aa27ffffffff0100c2eb0b000000001976a9145ac3b102cf5c113cded985d1216607524190ab218
    8ac0400473044022039a1a3b25dc3205b0451258a86a2e89f03f76c0879713c076df710eab5e6
    307f0220513337db9380ecd253daf70203fdb4ed4df2a57062be7a236c8ab8bd209001cc01483
    045022100e18cc428937baea32a8d1ffa79ad088aaac2465c464a1d8201582576c543f5e02204
    e3266de09415206f7ca1125333d0c17c8e39b30ae703295b5eccc31a7c8bdda0169522102e4ac
    a8f880bba36a54d87759d4fd516a20752e60d48eb67ce325907aa51dc4992102c03534156f845
    e19943c74be436cbdde10faabee67b54848dd4c29bf4ffc7fcf2103774aaaba0b572620af53df9
    9cac5d720487d90a3ee7017de720ff3efe406bf1b53ae00000000",
```

```

    "txid": "6cf13cee271e49db5e97546b91c8c48bd0b928c137dfe4b734bc7e405c98ab7d",
    "hash": "2cf69c3275c0598c9bc7a50e1512f198bea51fafaab0ca7fa26c7a0d3fe7abb3",
    "size": 375,
    "vsize": 184,
    "version": 1,
    "locktime": 0,
    "vin": [
      {
        "txid": "0ae8805cea766dafa5efeb126d98fbc7bf9c98b01666542e639f982562872c3",
        "vout": 0,
        "scriptSig": {
          "asm":
            "0020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58faa27",
          "hex":
            "220020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58faa27"
        },
        "txinwitness": [
          "",
          "3044022039a1a3b25dc3205b0451258a86a2e89f03f76c0879713c076df710eab5e6307f0220513337db9380ecd253daf70203fdb4ed4df2a57062be7a236c8ab8bd209001cc01",
          "3045022100e18cc428937baea32a8d1ffaf79ad088aaac2465c464a1d8201582576c543f5e02204e3266de09415206f7ca1125333d0c17c8e39b30ae703295b5eccc31a7c8bdda01",
          "522102e4aca8f880bba36a54d87759d4fd516a20752e60d48eb67ce325907aa51dc4992102c03534156f845e19943c74be436cbdde10faabee67b54848dd4c29bf4ffc7fcf2103774aaaba0b572620af53df99cac5d720487d90a3ee7017de720ff3efe406bf1b53ae"
        ],
        "sequence": 4294967295
      }
    ],
    "vout": [
      {
        "value": 2.00000000,
        "n": 0,
        "scriptPubKey": {
          "asm": "OP_DUP OP_HASH160 5ac3b102cf5c113cded985d1216607524190ab21 OP_EQUALVERIFY OP_CHECKSIG",
          "hex": "76a9145ac3b102cf5c113cded985d1216607524190ab2188ac",
          "reqSigs": 1,
          "type": "pubkeyhash",
          "addresses": [
            "monsYCVjaTxcnHyPaPXH5a9Sf7pE4b3iw6"
          ]
        }
      }
    ],
    "blockhash": "00000000000009bb7cb4e725a7993563546671896af42fa6d4952686ce38a17",

```

```
    "confirmations": 156944,  
    "time": 1473269800,  
    "blocktime": 1473269800  
}
```

We will be analyzing this output again after a few pages.

The command `getrawtransaction` is executed on the testnet network. The transaction hash starting with 6cf1 is unique. The inputs have an extra field called `txinwitness`.

The question is, how did we arrive at this transaction? As always, we have a series of `printf` functions written in the Bitcoin source code. These `printf`s are displayed when the Bitcoin server meets a transaction created using the rules of Segregated Witness. This is one of the smallest segregated witness transaction you will ever see in your life.

This same transaction in the testnet Bitcoin explorer (<http://tbtc.blockr.io/>) does not show the `txinwitness` field. This is the magic of Segregated Witness. The older Bitcoin clients do not understand a Segregated Witness, for them a Segregated Witness transaction is an odd transaction but all the same, correct, follows the rules. As we have always been saying, follows the BIP's.

Let's understand at least one basic idea of Segregated Witness. The first hex bytes of transaction are displayed as.

```
01 00 00 00  
00 01  
01 c372285682f939
```

The first four bytes are the transaction version. Even though, it is a Segregated Witness transaction, the version number does not matter at all. It yet remains at 1.

The fifth byte is the start of the number of inputs. C++ has a unique way of handling arrays or vectors. They always start with a number, which is a count of the number of inputs present in a transaction.

Till now, we believed that the following was Gospel; every transaction had to have at least one input and one output. Here the input size of 0. It is against the rule book, there cannot ever be a transaction with 0 inputs. Where would the money come from? Even Coinbase transactions have an input, though not used.

This is the brilliance of the Bitcoin developers or of a soft fork. If the input size is zero, then it is ignored. But now the next byte is called the flags byte which will be 1. Following this byte is the real count of inputs, 01, followed by the transaction hash that brings in inputs starting with c372. The `txid` field of the input or `vin` confirms the same. The transaction hash value is reversed and shown.

There are two transaction hashes, `txid` and `hash`. The `txid` field has the same value as before, the `hash` field is a new hash value.

Let's explain what Segregated Witness really means. The word "Segregate" means separate. The word "Witness" is someone who can vouch for something that has taken place. Here the "Witness" is the signature bytes that proves or validates the actual owner. The signature bytes are also called Witness. They are not placed in the input anymore where they are normally found, the field called `scriptSigs`, but on the outside. The transaction id is now calculated without the signature and hence transaction malleability is thrown out with the bathwater. More on this later.

Transaction malleability once again, is basically replacing the opcodes in the `scriptPubKey` with some other opcodes to perform the same task, but the transaction hash changes. If the transaction hash is computed by not using the signature bytes, the hash would always remain the same. Again, because all the bytes that go in computing this new hash are constant data like the public key.

As a result, there are two sizes now, the old one and the new size or size and vsize which is calculated by giving different weights to bytes. The field size is larger than the field vsize, it takes into account all the bytes, the newer field vsize does not take the witness data into account.

The txinwitness field is basically signatures. Let's break it up, including the last multi-sig. All this we did in an earlier chapter, nothing changes.

```

30 — DER
44 — length 68
022039a1a3b25dc3205b0451258a86a2e89f03f76c0879713c076df710eab5e6307f
0220513337db9380ecd253daf70203fdb4ed4df2a57062be7a236c8ab8bd209001cc
01 — SIGHASH_ALL

52 — OP_2
21 — Length 33
02e4aca8f880bba36a54d87759d4fd516a20752e60d48eb67ce325907aa51dc499
21 — Length 33
02c03534156f845e19943c74be436cbdde10faabee67b54848dd4c29bf4ffc7fcf
21 — Length 33
03774aaaba0b572620af53df99cac5d720487d90a3ee7017de720ff3efe406bf1b
53 — OP_3
ae — OP_CHECKMULTISIG

```

Now let's do something different. We downloaded three different Bitcoin copies using the bitcoind server version 0.11 and 0.12 and 0.13. Then we simply named the command bitcoind as bitcoind12 and bitcoind11. Run the same command in different bitcoind copies. The content will change depending upon the Bitcoin version used. We are crazy to have so many Bitcoin copies downloaded.

The client software bitcoin-cli has very little to do. The actual work is done by the server bitcoind and not the client. All three bitcoin servers download the same blockchain database or what we thought. We were wrong.

```
$bitcoin-cli11 getrawtransaction
```

Output

```

6cf13cee271e49db5e97546b91c8c48bd0b928c137dfe4b734bc7e405c98ab7d 1
{
  "hex" :
    "0100000001c372285682f939e6426566018bc9f97bbc8fd926b1fe5eaaaf6d76ea5c80e80a000
    0000023220020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58faa27f
    fffffff0100c2eb0b000000001976a9145ac3b102cf5c113cded985d1216607524190ab2188ac00
    000000",
  "txid" : "6cf13cee271e49db5e97546b91c8c48bd0b928c137dfe4b734bc7e405c98ab7d",
  "version" : 1,
  "locktime" : 0,
  "vin" : [
    {
      "txid" : "0ae8805cea766dafaa5efeb126d98fbc7bf9c98b01666542e639f982562872c3",
      "vout" : 0,
      "scriptSig" : {
        "asm" :

```

```
        "0020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58faa27",
        "hex" :
        "220020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58faa27"
    },
    "sequence" : 4294967295
}
],
"vout" : [
{
    "value" : 2.00000000,
    "n" : 0,
    "scriptPubKey" : {
        "asm" : "OP_DUP OP_HASH160 5ac3b102cf5c113cded985d1216607524190ab21
OP_EQUALVERIFY OP_CHECKSIG",
        "hex" : "76a9145ac3b102cf5c113cded985d1216607524190ab2188ac",
        "reqSigs" : 1,
        "type" : "pubkeyhash",
        "addresses" : [
            "monsYCVjaTxcnHyPaPXH5a9Sf7pE4b3iw6"
        ]
    }
}
],
"blockhash" : "000000000000009bb7cb4e725a7993563546671896af42fa6d4952686ce38a17",
"confirmations" : 160358,
"time" : 1473269800,
"blocktime" : 1473269800
}
```

There are no witnesses in sight. All the extra witness data was simply ignored by the earlier version. This is a soft fork. Earlier versions will yet work as advertised. They will not break or hang or complain, they simply do not care.

Let's look at the initial hex bytes again. First, they are fewer in size.

```
01 00 00 00 Transaction version
01 Number of inputs
c372 transaction hash getting Bitcoins.
```

For some strange reason, the initial bytes, 00 01, are not present on disk. This only concludes that when a Segregated Witness transaction arrives at an older version of the bitcoin server, a very different set of bytes are transmitted back and forth. This also means that when a miner sends you bytes he/she first checks what version of Bitcoin core you are running and then decides what bytes to send you.

Working With Different Index Folders for Versions 0.11 and 0.12 and 0.13

```
ch3502.py
import leveldb
import time
from cfuncs import *
```

```

def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def index(ans):
    (nVersion , offset) = base128(ans , 0)
    (nHeight, offset) = base128(ans , offset)
    (nStatus, offset) = base128(ans, offset)
    (nTx, offset) = base128(ans, offset)
    (nFile, offset) = base128(ans, offset)
    (nDataPos, offset) = base128(ans, offset)
    (nUndoPos, offset) = base128(ans, offset)
    (version,prevblockhash , merklehash , creationtime , difficulty , nonce) =
    struct.unpack("I32s32sIII" , ans[offset : offset + 80 ])
    #print time.ctime(creationtime)
    return(nFile , nDataPos)

def show(s):
    hash = "000000000000009bb7cb4e725a7993563546671896af42fa6d4952686ce38a17"
    hash = "62" + reversehash(hash)
    hash = hash.decode('hex')
    db = leveldb.LevelDB('/Users/vijaymukhi/Desktop/index' + s)
    ans = db.Get(hash)
    (nFile , nDataPos) = index(ans)
    #print "File Number %d nDataPos %d" % (nFile , nDataPos)
    f = open("/Users/vijaymukhi/Library/Application Support/Bitcoin%s/testnet3/
    blocks/blk%05d.dat" % (s , nFile ) , "r")
    f.seek(nDataPos - 8)
    magic = rbytes(f, 4)
    #print "Magic Number is %s" % magic.encode('hex')
    size = rint(f)
    header = f.read(80)
    (ver , prevhash , merklehash , time1 , diff , nonce) = struct.unpack("I32s32sIII" , header[0:80])
    #print "Previous Hash %s" % reversehash(prevhash.encode('hex'))
    #print "Merkle Hash %s" % reversehash(merklehash.encode('hex'))
    #print "Time %s:%d" % (time.ctime(time1), time1)
    #print "Bits %x" % (diff)
    #print "Nonce %d:%x" % (nonce , nonce)
    #print "File Pointer after block header is at %d" % f.tell()

```

```
notran = rvarint(f)
#print "Number of Transactions are %d" % notran
byte = f.read(50)
print "%s" % byte.encode('hex')

show("11")
show("12")
show("13")
```

Output

```
01000000010000000000000000000000000000000000000000000000000000000000000000000000ffff
ffff2f0304260e00042850
01000000010000000000000000000000000000000000000000000000000000000000000000000000ffff
ffff2f0304260e00042850
0100000000010100000000000000000000000000000000000000000000000000000000000000000000
0ffffff2f0304260e0004
```

To run this program, create three folders on the Desktop. Make sure they contain the index folders of Bitcoin version 0.11 and 0.12 and 0.13. Call these folders index11, etc. The show function simply passes the folder suffix and then uses the index chapter code to jump to a block hash starting with lots of 0's and then a 9bb7.

This is the block hash of the block number 927236 in the testnet network. This block contains our transaction starting with 6cf1. Check this out in your testnet browser.

To play safe, the block is read from the .dat file, just to prove that the data stored is very different from version 0.11 and 0.12 on one side and version 0.13 on the other.

We were surprised, all this time we thought that all versions of Bitcoin had the same data. Such is not the case. Therefore, running a bitcoind of version 11 on data downloaded by version 13 may not throw an error but the data carried is very different.

Back to Segregated Witnesses.

Let's us break up the initial transaction bytes returned by the getrawtransaction command manually. We have shown you the output some few pages earlier. A good idea is to make a copy of those bytes someplace and then check with what we are doing. The first bytes are the version field as

```
01000000
count 4

00 01
```

The count is 4 bytes or 8 characters. Then are the two bytes of the witness data, a 00 and 01, we ignore it. This is followed by the transaction hash, which is a vector 32 bytes large. A vector is contiguous series of bytes.

```
c372285682f939e6426566018bc9f97bbc8fd926b1fe5eaaaf6d76ea5c80e80a
count 4 + 32 + 1 = 37
```

Generally, one extra byte is added for a string because we need to account for the total number of vectors, not the size of each individual vector. There can be a 1000 vectors placed one after the other. The Compact format used by the Bitcoin Core team leave one byte to represent a number up to a value of 253. Numbers beyond this value need more bytes, This count accounts for the extra byte. It is a virtual count of bytes not a physical count. The 1 we added represents the total number of inputs.

A more technical explanation. The two fields, the transaction hash and the output index are represented as one vector or object. There will be multiples of such vectors instead of only one vector. That's why whenever an input or an output starts, one byte or more is required to give a count on the inputs or outputs out there.

If this number is less than 253 than one byte suffices to store this value. Here, count is nSize. This variable nSize is not the figment of our imagination but the same variable name in the source code.

The source code is written in such a way that the nSize variable, which stores the final size can be changed in two ways. The first way to update the value in this variable is by counting the actual bytes. The source code updates the nSize variable. Our tried and tested printf's in the source code disclose the actual bytes responsible for the size update. The second way is by using a seek function that does not move a file pointer here but simply updates the same nSize variable. Both have the same effect.

Once again, we expect you to be reading the source code. If not, ignore all references to the source code.

It must be noted that whenever we use the word seek, we are referring to the number of inputs or outputs or the length of a string that change the nSize variable. In the end, it is the nSize variable that stores the running size of the object. These numbers are C++ artefacts, they are present in the blockchain but not visible. More like the greasing oil that make our engines work.

The next 4 bytes are the output index of the transaction hash bringing in Bitcoins. It is a normal integer, so we leave aside 4 bytes, no more no less.

```
00000000
count=37 + 4 = 41
```

Then comes the public key minus the signature whose length is 35 bytes or 70 characters. Once again, since it is a vector, we add one more byte for the length at the start. The length byte 23 hex or 35 is the length of the string, in this case it is only the public key.

```
220020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58faa27
count = 35 + 1 = 36 + 41 = 77
Ffffffff
```

Finally, to end the inputs we have 4 bytes of the sequence number which are all F's.

```
Count = 77 + 4 = 81
```

This ends all the inputs.

Now let's get a count of the outputs. We may have only 1 output but that is not the reason for adding 1 to the size variable. It is for the space needed to store the number of outputs. To count the outputs, we start with the 8-byte bitcoin value stored as a vector.

```
00c2eb0b00000000
Count = 81 + 8 + 1 = 90
```

Then comes the script public key, the initial length byte 0x19 or 25 bytes. This is a string and the 1 is again for the length of the string.

```
76a9145ac3b102cf5c113cded985d1216607524190ab2188ac
Count 90 + 25 + 1 = 116
```

We end with the locktime, all 0's.

A function, `GetTransactionWeight` gets called in the source code. This function is in the header file, `validation.h` in the consensus folder. In this function, the `GetSerializeSize` function is called twice. The third parameter of this function is most important. When the function is called with a macro called `PROTOCOL_VERSION`, it returns the actual byte count of 375 bytes. This is the value of the size field. But, when we bitwise OR the above macro with another macro called `SERIALIZE_TRANSACTION_NO_WITNESS` then the `GetSerializeSize` function returns a value of 120 bytes. In other words, we have two different transaction types, one with witness data and one without witness data.

It unfortunately does not stop there, the original size of the transaction 375 is added to the newly minted value, 360 thus computing a value of 735. This value is the transaction weight and not the virtual size. But more trouble ahead.

What's annoying is that after this strenuous mental exercise, Pieter Wuille, in reply to a question on the website stackexchange, stated that the field `vsize` is for human consumption only. It is not used as of now by the Bitcoin Source code. But sometime later, it will be used to calculate the payable fees to the miners. Its more to compare apples with apples and not oranges.

The line shown above is a comment from the Bitcoin Source version 0.14. It is still not clear why there is a virtual seek of the size count by 1. There is a class called CSizeComputer to keep track of sizes of all objects.

[illegible]

```

100c2eb0b000000001976a9145ac3b102cf5c113cded985d1216607524190ab2188ac0400473
044022039a1a3b25dc3205b0451258a86a2e89f03f76c0879713c076df710eab5e6307f0220513
337db9380ecd253daf70203fdb4ed4df2a57062be7a236c8ab8bd209001cc01483045022100e1
8cc428937baea32a8d1ffaf79ad088aaac2465c464a1d8201582576c543f5e02204e3266de0941
5206f7ca1125333d0c17c8e39b30ae703295b5eccc31a7c8bdda0169522102e4aca8f880bba36
a54d87759d4fd516a20752e60d48eb67ce325907aa51dc4992102c03534156f845e19943c74be
436cbdde10faabee67b54848dd4c29bf4ffc7fc2103774aaaba0b572620af53df99cac5d720487
d90a3ee7017de720ff3efe406bf1b53ae00000000

```

These are the raw bytes of the same block that contains our segregated witness transaction.

```

bitcoin-cli getblock 000000000000009bb7cb4e725a7993563546671896af42fa6d4952686ce38a17 2
{
  "hash": "000000000000009bb7cb4e725a7993563546671896af42fa6d4952686ce38a17",
  "confirmations": 268618,
  "strippedsize": 430,
  "size": 721,
  "weight": 2011,
  "height": 927236,
  "version": 536870912,
  "versionHex": "20000000",
  "merkleroot":
    "bb04102b52b558b35e68601a7d42b40daa2f85b3f399b4b99d7c10800b44ec93",
  "tx": [
    {
      "txid": "5431e922b1b3b1577282f9479ed534f474738a279a6efdd7e279fd3e9367a75e",
      "hash": "af37638b22bdb7d94261fe919cb088a50ade68211dfa9abd00605f0e601a5018",
      "version": 1,
      "size": 265,
      "vsize": 238,
      "locktime": 0,
      "vin": [
        {
          "coinbase":
            "0304260e00042850d05704cb5b46110c94e1cf57a535000000000000122f4e6
            96e6a61506f6f6c2f5345475749542f",
          "sequence": 4294967295
        }
      ],
      "vout": [
        {
          "value": 3.72500000,
          "n": 0,
          "scriptPubKey": {
            "asm":

```

[illegible]


```

    "version": 1,
    "size": 375,
    "vsize": 184,
    "locktime": 0,
    "vin": [
      {
        "txid":
          "0ae8805cea766dafa5efeb126d98fbc7bf9c98b01666542e639f982562872c3",
        "vout": 0,
        "scriptSig": {
          "asm":
            "0020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58faa27",
          "hex":
            "220020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58faa27"
        },
        "txinwitness": [
          "",
          "3044022039a1a3b25dc3205b0451258a86a2e89f03f76c0879713c076df710eab5e6307f02205
            13337db9380ecd253daf70203fdb4ed4df2a57062be7a236c8ab8bd209001cc01",
          "3045022100e18cc428937baea32a8d1ffaf79ad088aaac2465c464a1d8201582576c543f5e022
            04e3266de09415206f7ca1125333d0c17c8e39b30ae703295b5eccc31a7c8bdda01",
          "522102e4aca8f880bba36a54d87759d4fd516a20752e60d48eb67ce325907aa51dc4992102c0
            3534156f845e19943c74be436cbdde10faabee67b54848dd4c29bf4ffc7fcf2103774aaaba0b57
            2620af53df99cac5d720487d90a3ee7017de720ff3efe406bf1b53ae"
        ],
        "sequence": 4294967295
      }
    ],
    "vout": [
      {
        "value": 2.00000000,
        "n": 0,
        "scriptPubKey": {
          "asm": "OP_DUP OP_HASH160 5ac3b102cf5c113cded985d1216607524190ab21
            OP_EQUALVERIFY OP_CHECKSIG",
          "hex": "76a9145ac3b102cf5c113cded985d1216607524190ab2188ac",
          "reqSigs": 1,
          "type": "pubkeyhash",
          "addresses": [
            "monsYCVjaTxcnHyPaPXH5a9Sf7pE4b3iw6"
          ]
        }
      }
    ],
    "hex":
      "01000000000101c372285682f939e6426566018bc9f97bbc8fd926b1fe5eaaaf6d76ea5c80e80
        a0000000023220020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58f

```

For some reason, the magic number and the block size are not considered to be a part of a block, they are pariahs. Let's now understand how the size and stripped size fields get their value. The size field is simply the size of the raw bytes, 721 bytes. The strippedsize is another kettle of fish. A block starts with the 6 field 80-bytes block header. The first 4 bytes are the version bytes.

The next 32 bytes are the previous block hash.

This hash is displayed backwards.

This is followed by the 4 bytes of the time. The block output shows us the time as 1473269800 or 0x57D05028.

After the time, comes the 4-bytes of the difficulty bits. The block output reverses it.

Finally, the last field of 4 bytes, the nonce.

The hashes have no seek of 1 byte added to the size.

The transactions data starts with a count of the number of transactions.

We start with the transaction version bytes, size 4.

The witness bytes 0001 are not needed for the size calculations.

Count = 86

As there is no real transaction hash we are referring to, the output index is all F's. In the world of Bitcoin Core, a value of all F's is an invalid value. Here, the output index has no meaning.

The 2f is the length of the signature and public key. Therefore, the digits or characters following are 94.

The getblock output calls the next set of bytes as Coinbase. We have already explained these bytes in detail earlier, but as we forget so do you. As we are dealing with version 2 transactions, the first byte will always be 03 as the block height is being stored in the next three bytes. The block height is 927236 in decimal and 0x0e 26 04 in hex. Reverse the bytes and we get 04260e.

The last field of the inputs the sequence number, 4 bytes of all F's

ffffff

Count = 170 + 4 = 174

This ends the first transaction's inputs.

Now comes the outputs. We have 3 outputs in all, which comes as a surprise as in the past we had only one output. Common sense dictates that the miner block reward should go to one output only. However, Bitcoin Core feels that even the miner rewards on paper can be shared and hence there are multiple outputs. This is where the count value is incremented by 1.

We will see later that these two extra outputs carry no money with them.

Count = 175

The amount in Bitcoins, 3.725 is now visible. A total of 8 bytes. Remember we are on the Testnet and not mainnet which explain the measly block reward.

20e6331600000000

count = 175 + 8 = 183

This is followed by the length script public key, we have the size in hex 19.

Count = 184

The 25 bytes of the first output's script public key. This miner receives the rewards.

76a914876fbb82ec05caa6af7a3b5e5a983aae6c6cc6d688ac

count = 184 + 25 = 209

This ends the first output, time to understand the second and third output.

There are 8 bytes filled with all 0's. This is the Bitcoin value. You can follow up with the getblock output to understand it better. A real output will not have an output of all 0's. This really has no meaning as the output generally tell us where the Bitcoins are going, plus there must be some Bitcoins to start with. 0 Bitcoins are going nowhere.

0000000000000000

count = 209 + 8 = 217

We have 0x26 or 38 bytes following. The count is increased by 1, thanks to the length bytes.

Count = 218

Now we have the actual Script Public Key field. The opcode 6a means OP_RETURN. The number 0x24 or 36 indicate that the next 72 digits are owned by the OP_RETURN opcode. Once again, an opcode 6a means that all is over. The next byte 0x24 or decimal 36 refers to the number of bytes following.

The bytes aa21a9ed is a magic or constant number and the remaining 32 bytes are for a hash value. We intend to compute this hash value ourselves later.

6a24aa21a9ed9c3bba9bf58abbc5d3667100786cfb60d0540aba6d6466e0edb2639a3e2d557b

count 218 + 38 = 256

This ends the second output, so let's start with the third one. Once again, the next 8 bytes of the Bitcoin value are all 0's. The output type is nulldata.

0000000000000000


```
nooutputs = '01'
bitcoinvalue = '00c2eb0b00000000'
lenscriptpublickey = '19'
scriptpublickey = '76a9145ac3b102cf5c113cded985d1216607524190ab2188ac'
witnessdata =
'0400473044022039a1a3b25dc3205b0451258a86a2e89f03f76c0879713c076df710eab5e6307
f0220513337db9380ecd253daf70203fdb4ed4df2a57062be7a236c8ab8bd209001cc01483045
022100e18cc428937baea32a8d1ffaf79ad088aaac2465c464a1d8201582576c543f5e02204e32
66de09415206f7ca1125333d0c17c8e39b30ae703295b5eccc31a7c8bdda0169522102e4aca8f
880bba36a54d87759d4fd516a20752e60d48eb67ce325907aa51dc4992102c03534156f845e19
943c74be436cbdde10faabee67b54848dd4c29bf4ffc7fcf2103774aaaba0b572620af53df99ca
c5d720487d90a3ee7017de720ff3efe406bf1b53ae'locktime = "00000000"

s = version + dummy + flags + noinputs + thash + outputindex + lenscriptSig + scriptSig +
sequencenumber + nooutputs + bitcoinvalue + lenscriptpublickey + scriptpublickey + witnessdata +
locktime
s = s.decode('hex')
s = chash(s)
if "2cf69c3275c0598cfbc7a50e1512f198bea51fafaab0ca7fa26c7a0d3fe7abb3" ==
reversehash(s.encode('hex')):
    print "ok"
else:
    print "Vijay Mukhi is a embecile"

s = version + noinputs + thash + outputindex + lenscriptSig + scriptSig + sequencenumber +
nooutputs + bitcoinvalue + lenscriptpublickey + scriptpublickey + locktime
s = s.decode('hex')
s = chash(s)
if "6cf13cee271e49db5e97546b91c8c48bd0b928c137dfe4b734bc7e405c98ab7d" ==
reversehash(s.encode('hex')):
    print "ok"
else:
    print "Vijay Mukhi is a embecile"
```

Output

```
ok
ok
```

The output of the getrawtransaction command shows two fields, txid and hash. It is shown below:

```
"txid": "6cf13cee271e49db5e97546b91c8c48bd0b928c137dfe4b734bc7e405c98ab7d",
"hash": "2cf69c3275c0598cfbc7a50e1512f198bea51fafaab0ca7fa26c7a0d3fe7abb3",
```

Let's now find out how these values have been calculated. Basically, it takes all the bytes of the transaction including the newer witness data and calculates the hash value. This value must match the value in the hash field. So, the version, dummy, flags up to the witness data and locktime are concatenated and then the hash value is calculated.

The txid hash uses the same fields as seen above, minus the dummy, flags and witness data. The transaction hash will never ever change as the signatures are removed. One more example of transaction malleability.

Let's now figure out the hash value after the OP_RETURN. We have two transactions, one is a Coinbase transaction and the other transaction is with the following txid hash value

6cf13cee271e49db5e97546b91c8c48bd0b928c137dfe4b734bc7e405c98ab7d

```
ch3504.py
import hashlib
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def hash2( a , b):
    c = a + b
    hashtemp = hashlib.sha256(c).digest()
    hashfinal = hashlib.sha256(hashtemp).digest()
    return hashfinal
#927236 and 927237 927241 927242 927244 are segwit
coinbase =
"0000000000000000000000000000000000000000000000000000000000000000".decode('hex')
hash =
"2cf69c3275c0598cfbc7a50e1512f198bea51fafaab0ca7fa26c7a0d3fe7abb3".decode('hex')
#transaction hash of the witness transaction
rhash =
"b3abe73f0d7a6ca27fcab0aaaf1fa5be98f112150ea5c7fb8c59c075329cf62c".decode('hex')
#this is hash reversed
hashf = hash2(coinbase , rhash)
firsthash = hashf.encode('hex')
print firsthash
hashf = hash2(firsthash.decode('hex') , coinbase)
print hashf.encode('hex')
12c2e70ea4880ab28a60a9e105eabd78bd2186a58c7792aa5d28dfda8ffdbd4e
9c3bba9bf58abb5d3667100786cfb60d0540aba6d6466e0edb2639a3e2d557b
```

A Coinbase transaction hash is all 0's. The hash field unlike the txid field, is the hash of all the transaction bytes. This includes the witness data. If the witness data is ignored in a hash, then anyone can change this data and there will be no hash monitoring it.

The objective here is to calculate the hash after the OP_RETURN opcode and confirm that we refer to this hash. That's why we need a working example or a peek at the source code. Ours is one of the simplest case where there are only two transaction hashes. We have already learnt about calculating the Merkle hash using recursion in one of the earlier chapters.

The function hash2 has been copied from the Merkle hash chapter. A Merkle hash is calculated of the Coinbase and the txid, but the bytes are reversed. The result is a temporary hash or a witness hash.

Then, once again a hash is calculated of this value and a nonce, which is all 0's. The newly calculated hash is stored in the second output of the Coinbase transaction. This hash is similar to the one seen in the outputs.

```
ch3505.py
import hashlib
import subprocess
```

```
import json
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def hash2( a , b):
    c = a + b
    hashtemp = hashlib.sha256(c).digest()
    hashfinal = hashlib.sha256(hashtemp).digest()
    return hashfinal

raw = subprocess.check_output(["bitcoin-cli" , "getblockhash" , "927236" ])
raw = subprocess.check_output(["bitcoin-cli" , "getblock" , raw , "2"])
raw = json.loads(raw)
hash = raw['tx'][1]['hash']
hash = reversehash(hash)
coinbase =
"0000000000000000000000000000000000000000000000000000000000000000".decode('hex')
hash = hash.decode('hex')
hashf = hash2(coinbase , hash)
firsthash = hashf.encode('hex')
hashf = hash2(firsthash.decode('hex') , coinbase)
print hashf.encode('hex')
output = raw['tx'][0]['vout'][1]['scriptPubKey']['hex'][12:]
print output
```

Output

```
9c3bba9bf58abb5d3667100786cfb60d0540aba6d6466e0edb2639a3e2d557b
9c3bba9bf58abb5d3667100786cfb60d0540aba6d6466e0edb2639a3e2d557b
```

This program is similar to the earlier code but it is more generic. We use the same code that give us a Python dictionary, tx from where the complete transaction hash is extracted. From the tx list, the Coinbase transaction is skipped and the second transaction is read with an offset of 1. Then the hash field is accessed. The rest of the code is the same as before.

The following fields are accessed: The Coinbase transaction, which has an offset index of 0 in the tx list, then the vout list, the Merkle hash, which is the first output, then the scriptPubKey dictionary and finally the hex key. The source code ensures that the magic number aa21a9ed is present. Both hashes are the same.

Now let's make our code more generic.

ch3506.py

```
import hashlib
import subprocess
import json
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))
def hash2( a , b):
    c = a + b
    hashtemp = hashlib.sha256(c).digest()
    hashfinal = hashlib.sha256(hashtemp).digest()
```



```

return hashfinal
for blockno in ["927236" , "927242" ]:
    raw = subprocess.check_output(["bitcoin-cli" , "getblockhash" , blockno ])
    raw = subprocess.check_output(["bitcoin-cli" , "getblock" , raw , "2"])
    raw = json.loads(raw)
    hash = raw['tx'][1]['hash']
    hash = reversehash(hash)
    coinbase =
    "0000000000000000000000000000000000000000000000000000000000000000".decode('hex')
    hash = hash.decode('hex')
    hashf = hash2(coinbase , hash)
    firsthash = hashf.encode('hex')
    hashf = hash2(firsthash.decode('hex') , coinbase)
    output = raw['tx'][0]['vout'][1]['scriptPubKey']['hex'][12:]
    if output == hashf.encode('hex'):
        print "Block number %s is good" % blockno
    else:
        print "Error is Block Number %s" % blockno

```

Output

Block number 927236 is good

Block number 927242 is good

We have a list of strings that double up as two block numbers with segregated witness transactions. These two blocks are chosen on purpose, as they have only two transactions, the unavoidable Coinbase transaction and a single segregated witness transaction.

We read the source code to figure out the Merkle hash and nonce. The only drawback here is that a block is not created every 10 seconds but every 10 minutes on an average.

We downloaded the testnet up to block 927200. We made multiple copies of this folder. Each time we ran bitcoind, it started downloading blocks from 927200. The printf's placed in the source code get called on block number 927236. We then stop bitcoind.

You may be wondering as in why did we not create our own Segregated Witness transaction and use it to understand segregated witnesses, as before.

We first created two testnet addresses using the command `getnewaddress`.

```

bitcoin-cli getnewaddress
mksnPvsUg7zSEHu8G6mEvrWwCSyRqbvkYp
bitcoin-cli getnewaddress
msUSiBsrvT11t7mp77u258QzTvD3Tep3XH

```

We then created a segwit address using the command `addwitnessaddress`.

```

bitcoin-cli addwitnessaddress msUSiBsrvT11t7mp77u258QzTvD3Tep3XH
2N3cn9NKg3S1FFV7FFQNZwWYVdnPkmYzP5f

```

Now to add actual Bitcoins to this Bitcoin address starting 2N.

We used the following 3 faucets for this task. They may not work when you read this book.

<https://testnet.manu.backend.hamburg/faucet>

<https://faucet.haskoin.com>

<https://testnet.coinfaucet.eu/en/>

An account in the Bitcoin wallet is simply a name with some money and some addresses associated with it. We create an account name, vijay and if you remember there is a default account called "". Is this a real name as it cannot be empty!!! We also add some testcoins from the above faucets.

```
bitcoin-cli setaccount 2N3cn9NKg3S1FFV7FFQNZwWYVdnPkmYzP5f vijay
```

```
bitcoin-cli getaccount 2N3cn9NKg3S1FFV7FFQNZwWYVdnPkmYzP5f  
vijay
```

Now we check the balances in our accounts, after a wait.

```
bitcoin-cli listaccounts  
{  
  "": 7.50047473,  
  "vijay": 15.25100757  
}
```

Both accounts, "" and vijay have enough money. We now create a segwit transaction by sending money to a random testnet address.

```
bitcoin-cli sendfrom vijay mksnPvsUg7zSEHu8G6mEvrWwCSyRqbvkYp 0.99  
c3aba003f47476206117c32a738fef3432be0e03952978483be939435874fcb7
```

The sendfrom command needs an account name, basically who should receive the Bitcoins, the Bitcoin address of the account name and finally a Bitcoin value. The data received is the transaction id of the transaction.

```
bitcoin-cli getrawtransaction  
c3aba003f47476206117c32a738fef3432be0e03952978483be939435874fcb7 1  
{  
  "txid": "c3aba003f47476206117c32a738fef3432be0e03952978483be939435874fcb7",  
  "hash": "fd87f0f865680e7087a257e02b026d462249d786e57a5a0c4a20e01b7d81890e",  
  "version": 2,  
  "size": 251,  
  "vsize": 170,  
  "locktime": 1195960,  
  "vin": [  
    {  
      "txid": "0bdadbf75336470a63649368507f0aa5e7b50fe3e471a0f59a2466278731a166",  
      "vout": 0,  
      "scriptSig": {  
        "asm": "00148327fb6a7747c7cd618e83d47cc64d0df7f9d148",  
        "hex": "1600148327fb6a7747c7cd618e83d47cc64d0df7f9d148"  
      },  
      "txinwitness": [  

```

```

        "304402200f14472e9c4dba9b8105f8bb1ef8b1c9e9b8c5a37f97f86a63a58a92d88fdd9202202
        7687b868b800287be2297f69aaf9cb88a055618ba48f9e1e5013005111febc501",
        "037b0c6db6c88032059cc1d877085dc88826212aa31f86d063a8fbe81f5d9356d1"
    ],
    "sequence": 4294967294
  }
],
"vout": [
  {
    "value": 0.99000000,
    "n": 0,
    "scriptPubKey": {
      "asm": "OP_DUP OP_HASH160 3ac919e1badbad52a66d100fdea7a8f3c775711b
      OP_EQUALVERIFY OP_CHECKSIG",
      "hex": "76a9143ac919e1badbad52a66d100fdea7a8f3c775711b88ac",
      "reqSigs": 1,
      "type": "pubkeyhash",
      "addresses": [
        "mksnPvsUg7zSEHu8G6mEvrWwCSyRqbvkYp"
      ]
    }
  },
  {
    "value": 0.30991481,
    "n": 1,
    "scriptPubKey": {
      "asm": "OP_DUP OP_HASH160 1028bfab3a51fd753f3a25c1542b6dda9360905b
      OP_EQUALVERIFY OP_CHECKSIG",
      "hex": "76a9141028bfab3a51fd753f3a25c1542b6dda9360905b88ac",
      "reqSigs": 1,
      "type": "pubkeyhash",
      "addresses": [
        "mgzPvPkYDbeggPAQmunzcgxRMkiAXxQr9C"
      ]
    }
  }
],
"hex":
"0200000000010166a131872766249af5a071e4e30fb5e7a50a7f50689364630a473653f7dbda0
b00000000171600148327fb6a7747c7cd618e83d47cc64d0df7f9d148feffffff02c09ee60500000
0001976a9143ac919e1badbad52a66d100fdea7a8f3c775711b88ac79e4d801000000001976a
9141028bfab3a51fd753f3a25c1542b6dda9360905b88ac0247304402200f14472e9c4dba9b81
05f8bb1ef8b1c9e9b8c5a37f97f86a63a58a92d88fdd92022027687b868b800287be2297f69aaf
9cb88a055618ba48f9e1e5013005111febc50121037b0c6db6c88032059cc1d877085dc888262
12aa31f86d063a8fbe81f5d9356d1b83f1200"
}

```

The txinwitness field is the one to look for. There are the two hashes as well, the txid and hash. Also, the dummy and flags variables are visible in the hex section. We waited for 2 days, but no one wanted to add our transaction to a block. Check this out yourself. This is the bane of testnet.

We used the command `settxfee 0.001` to increase the transaction fee payable to a miner.

```
bitcoin-cli sendfrom vijay mksnPvsUg7zSEHu8G6mEvrWwCSyRqbvkYp 0.99
2b95f90f7401721fc74a09f3f0273818928291c3e537b4fc4d6b52b0b12a26f5
```

This transaction and the previous transaction are both present in the same block number 1196023. Please figure out the block hash and see for yourself that these transactions are segwit transactions. Just for your information, this block has 1883 transactions.

We spent a lot of time finding blocks that have only two transactions, a Coinbase and a SegWit transaction. It's easier with 2 transactions. One of the many reasons why this book takes a long time to write.

We found only one Python library that understood Segregated Witness as of today, the 16th of September 2017. The following command installs this library.

```
sudo pip3 install chainside-btcpy
```

This library works only with Python3 and in Python3, that the `print` command is now a function. Old people hate changes. Run with `python3` and not `python`.

```
ch3507.py
from btcpy.setup import setup
setup('testnet')
python3 ch3707.py
```

This program simply choses one of the three networks, mainnet or regtest or testnet. It must be executed only once.

```
Ch3708.py
from btcpy.structs.transaction import Transaction
from btcpy.structs.block import Block
from btcpy.structs.crypto import PublicKey, PrivateKey
a =
"01000000000101c372285682f939e6426566018bc9f97bbc8fd926b1fe5eaaaf6d76ea5c80
80a0000000023220020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58f
aa27ffffffff0100c2eb0b000000001976a9145ac3b102cf5c113cded985d1216607524190ab218
8ac0400473044022039a1a3b25dc3205b0451258a86a2e89f03f76c0879713c076df710eab5e6
307f0220513337db9380ecd253daf70203fdb4ed4df2a57062be7a236c8ab8bd209001cc01483
045022100e18cc428937baea32a8d1ffaf79ad088aac2465c464a1d8201582576c543f5e02204
e3266de09415206f7ca1125333d0c17c8e39b30ae703295b5eccc31a7c8bdda0169522102e4ac
a8f880bba36a54d87759d4fd516a20752e60d48eb67ce325907aa51dc4992102c03534156f845
e19943c74be436cbdde10faabee67b54848dd4c29bf4ffc7fcf2103774aaaba0b572620af53df9
9cac5d720487d90a3ee7017de720ff3efe406bf1b53ae00000000"
tx = Transaction.unhexlify(a)
print(tx)
python3 ch3508.py
```

Output

```

SegWitTransaction(version=1,
ins=[TxIn(txid=0ae8805cea766dafaa5efeb126d98fbc7bf9c98b01666542e639f982562872c3,
txout=0,
script_sig=0020a4af1cc468ded44a25ce02175712b1646bbe2bac1f2801b755ec31dfb58faa27,
sequence=4294967295, witness=Witness(["",
"3044022039a1a3b25dc3205b0451258a86a2e89f03f76c0879713c076df710eab5e6307f02205
13337db9380ecd253daf70203fdb4ed4df2a57062be7a236c8ab8bd209001cc01",
"3045022100e18cc428937baea32a8d1ffaf79ad088aaac2465c464a1d8201582576c543f5e022
04e3266de09415206f7ca1125333d0c17c8e39b30ae703295b5eccc31a7c8bdda01",
"522102e4aca8f880bba36a54d87759d4fd516a20752e60d48eb67ce325907aa51dc4992102c0
3534156f845e19943c74be436cbdde10faabee67b54848dd4c29bf4ffc7fcf2103774aaaba0b57
2620af53df99cac5d720487d90a3ee7017de720ff3efe406bf1b53ae"])]),
outs=[TxOut(value=200000000, n=0, scriptPubKey='OP_DUP OP_HASH160
5ac3b102cf5c113cded985d1216607524190ab21 OP_EQUALVERIFY OP_CHECKSIG')],
locktime=Locktime(0))

```

We run the same command, `getrawtransaction` using the transaction hash starting with 6cf1. These bytes are seen in variable `a`. The function `unhexlify` recognizes this as a witness transaction so, it displays the witness field also. There are zillion things that you can do with this library.

Its far better to experiment with Python code that tinker with the C++ code.

We have lots and lots of unfinished business in this chapter but if we do not stop, this book will cross a 1000 pages.

What is more crucial now is to explain how a witness address is computed. The third Coinbase output speaks of Witness Commitments. Plus, the `txwitness` fields in the above transactions need more explanation.

Bitcoins look godly as it rests on the strong shoulders of Segregated Witnesses, so don't give up on it.

According to folklore, segregated witness was triggered at block number 470707. Similarly, block number 494784 triggered the segwit2x hard fork. On August 23, 2017, the segwit soft fork took place at block number 481824. We have not independently verified these block numbers. What we know is that this block will be mined sometime in November, not the exact date and time as mining blocks occurs at an average of 10 minutes. The block weight will increase to 8MB for segwit enabled clients and 2MB for non segwit clients.

There are only three compatible segwit clients today, `btcd` and `classic` and `unlimited`. The blog post very clearly says that Bitcoin Core is not compatible with the above releases.

Our take, the hard fork will split the Bitcoin Core Blockchain into 2 or more different chains and nobody will care, as SegWit is activated in both Blockchains. The Bitcoin Core developers are not sitting idle and they are talking of using Satellites for Bitcoin transactions and about thunderstorms and lighting. Other companies have already decided to use Bitcoin as a smart contract platform over Ethereum. Everyone has bet the barn on SegWit and Smart Contracts. So, have we.

But how do we figure out if a miner supports Segwit or for that matter, any other soft fork. Unfortunately, there is no field in the Bitcoin block that talks of soft or hard forks. All this time, we kept saying that the version field in the block header is of no use. Well someone, somewhere has been listening to us. The version field is actually doing something useful for some time.

Let's first create a SQL table as

```

create table blockdata1(blockno integer , blockhash varchar , ctime varchar, time1
integer , ver integer , ver1 varchar);

```

Some earlier code is copied in the program given below.

```
ch3509.py
from cfuncs import *
import psycopg2
import time
conn = psycopg2.connect("dbname='postgres' user='postgres' host='localhost' password='accord'")
cur = conn.cursor()
blockno = 0
for fno in range(0 ,10000):
    fname = "/Volumes/VijaySamsun/Bitcoin/blocks/blk%05d.dat" % fno
    try:
        f = open(fname , "rb")
        print "File %s:%d" % (fname,blockno)
    except:
        break
    while ( True ):
        try:
            where = f.tell()
            magic = rint(f)
            if magic == 0:
                continue
        except:
            break
        size = rint(f)
        header = f.read(80)
        blockhash = chash(header)
        (ver , prevhash , merklehash , time1 , diff , nonce ) = struct.unpack("I32s32sIII" , header)
        ver1 = "%08x" % ver
        s1 = "insert into blockdata1 values (%d,'%s','%s',%d,%d,'%s')" % (blockno ,
            blockhash[:-1].encode('hex') ,time.ctime(time1) , time1 , ver , ver1)
        cur.execute(s1)
        blockno = blockno + 1
        f.seek(size - 80, 1)
        conn.commit()
```

This program simply adds one SQL record for every block. The values stored are of the not-so-useful virtual block number, the very useful blockhash to uniquely identify a block, the block creation time in English as well as in seconds from a certain epoch, etc. What is most important for us is the version number in an integer format as well as a hex string.

```
select ver1 , count(*) from blockdata1 group by ver1 order by 2 desc;
00000001 | 215047
00000002 | 140752
20000000 | 51637
00000003 | 29304
00000004 | 27212
```

20000002		15787
20000001		4976
30000000		2058
20000012		1180
20000010		304
20000004		212
30000001		114
20000007		41
08000004		39
30000007		4

This SQL statement gives a count as in how many times the version number changed and how many blocks used that version number. What stands out is that we have version number 1,2 3 and 4 and then some of them starting with 0x200.

In the good old days, the version number changed when the Bitcoin blockchain rules changed. The problem here is that there can be only one change at a time and that change cannot be reused. Then there was bip 9 that says let's bring order to version numbering and it introduced version bits. Instead of looking at the version number as a one large integer, it is now looked as a series of bits.

The last three bits are 001 which in hex will show the last 4 bits or a nibble as 0x200. That's why you see the top version byte nibble as a 200 these days. This widens the scope as at some future day, the other top two bits can be used as version numbers. The remaining 29 bits can denote 29 simultaneous soft forks happening at the same time.

Let's give a specific example. The Bitcoin Core's Segwit fork decided that the second bit being 1 would denote that a miner is signaling support for Segwit. In other words, a miner simply sets bit 2 on, if he/she wants to announce to the world that he/she supports segwit. The only problem is that a miner sends a block out only when he gets the hash puzzle right, so if a miner never mined a correct block we will not know his/her preferences. This means that blocks with a version number 20000002 are supporters of segwit. But there are only 15000 blocks that support only segwit. Miners or peers are always talking to each other.

On the other hand, the competitor segwit2x uses bit 4 which means that miners that use version number 20000010 are segwit2x miners. As a miner, if you want to support both segwi2x and segwit, you use a version number 20000012.

In the past, another activation called csv or checksequenceverify had to have the 1st bit on. This means that the version number of 20000001 means csv or nlocktime support. Once these soft forks activations are done, then we revert to a version number of 20000000 which means no soft fork activations are being processed. In the same vein, supporting segwit2x and csv means a version number of 20000011 and there are none so far.

A search on Google shows that Bitcoin XT has a version of 20000007 and 2MB block a version number of 30000000. We understood versioning bits only after seeing the actual version bits. As there are no ongoing forks, the default version number of all recent blocks are 0x20000000.

This Url gives a list of all the official deployments

<https://github.com/bitcoin/bips/blob/master/bip-0009/assignments.mediawiki>.

CHAPTER 36

Bitcoin Core 0.15, UTXO and More

The new release of Bitcoin Core (version 0.15) focusses primarily on performance issues, making Bitcoin run faster. There are new features added as well. Today, the blockchain sitting on our machines is already 180 GB large and growing. With segregated Witnesses in place, the Bitcoin network will grow bigger and the only drawback will be the speed, we need more performance. Now, the code gets more difficult to understand. UTXOs are replaced with something much simpler so everyone can understand it. However, this newer UTXO format is not used in the Bitcoin Cash or the segwit2x implementations.

The scene is that some fork of bitcoin uses the older UTXO format and some use the newer one. So, there is confusion. The Bitcoin developer community ensured that every fork looks and feels different.

Let's understand the newer UTXO format using python code. It is the same code from the UTXO chapter, here only differences are explained.

```
ch3601.py
import leveldb
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstate")
cnt = cntc = cntC = cnto = cntb = cnte = 0
for k,v in db.RangeIter():
    cnt = cnt + 1
    if k[:1] == 'c' :
        cntc = cntc + 1
    if k[:1] == 'C' :
        cntC = cntC + 1
    elif k[:1] == 'B':
        cntb = cntb + 1
    elif ord(k[:1]) == 14:
        cnto = cnto + 1
    else:
        cnte = cnte + 1
    print "%c" % (k[:1])
```

Output

```
print "Total Number of keys %d" % cnt
print "Total Number of c keys %d" % cntc
print "Total Number of C keys %d" % cntC
print "Total Number of B key %d" % cntb
print "Total Number of obfuscate_key %d" % cnto
print "Total Number of extra keys %d" % cnte
```


Total Number of keys 51320344
 Total Number of c keys 0
 Total Number of C keys 51320342
 Total Number of B key 1
 Total Number of obfuscate_key 1
 Total Number of extra keys 0

In January 2017, we had reached a milestone of 17 million records. But in September 2017, there are 51 million plus records in the leveldb database. The size of the chainstate folder now is a mammoth 1.9GB, much larger than ever before. This only proves that the new UTXO dataset is by far bigger in size.

The earlier Coin key c is now replaced by a capital C. The last block key, B and the key called obfuscate_key see no change at all.

```

ch3602.py
import leveldb
def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)
def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n
def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
  
```

```
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
    if k == 8:
        k = 0
    return finalstr

def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)

def displaychainstate( finalstr , hash):
    finalstr = finalstr.decode('hex')
    print "Key Value %s" % (finalstr.encode('hex'))
    print "Key Hash %s" % hash.encode('hex')
    hash0 = ord(hash[0])
    hash = hash[1:]
    if len(hash) != 32:
        length = ord(hash[0]) - 0x80
        hash = hash[1:]
        hash0 = hash0 + 128 * (length + 1)
    print "Transaction Hash %s" % hash.encode('hex')
    print "Offset Index %d" % hash0
    (nHeight , offset) = base128(finalstr, 0)
    print "Block Height %d:%d offset %d" % (nHeight, nHeight / 2 , offset)
    print "Key Value %s" % finalstr[offset:].encode('hex')
    (bvalue, offset) = base128a(finalstr, offset)
    print "Bitcoin Value %d offset %d" % (bvalue, offset)
    print "Key Value %s" % finalstr[offset:].encode('hex')
    publickey = finalstr[offset:]
    print "Public Key %s:%d" % (publickey.encode('hex') , offset)

xorkey = ""
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstate")
for k,v in db.Rangelter():
    if k[:1] == 'C':
        finalstr = xorstring(v , xorkey)
        hash = k[1:]
        displaychainstate(finalstr , hash[:-1])
    break
```

```
elif ord(k[:1]) == 14:
    xorkey = v[1:]
```

Output

```
Key Value baa81480d0c157010baecb9d6a8a36d33e8332cb5f98ad1c86a6b6ee
Key Hash
007074402e5a9bf461acab0bb5d8c1ea48f626057b7a2a2a67bc8ddcc95a010000
Transaction Hash
7074402e5a9bf461acab0bb5d8c1ea48f626057b7a2a2a67bc8ddcc95a010000
Offset Index 0
Block Height 971924:485962 offset 3
Key Value 80d0c157010baecb9d6a8a36d33e8332cb5f98ad1c86a6b6ee
Bitcoin Value 381422 offset 7
Key Value 010baecb9d6a8a36d33e8332cb5f98ad1c86a6b6ee
Public Key 010baecb9d6a8a36d33e8332cb5f98ad1c86a6b6ee:7
```

We are displaying a key and value of a single leveldb record. The key starts with a C followed by a value that first represents an output index and then the 32-byte transaction hash. As before, numbers are varying in length, so we have no idea how many bytes make up the number. When the high bit is 0, the number ends.

In our case the first byte is 00, so the output index is 0. The bytes starting with 7074 belong to a transaction whose 0th output is not spent. Please verify this transaction hash in a blockchain explorer. The UTXO database has only unspent outputs. The new UTXO format is very simple, the key gives a transaction hash and enclosed within that are the unspent outputs. If a transaction hash has 5 unspent outputs, there will be 5 leveldb records and not one, like before. They will be unique because even if the transaction hash is the same, the starting output index is different.

The earlier UTXO format though very efficient, was a real pain in the neck, it was very difficult to understand, even code-wise. As there was only one transaction hash record with all unspent outputs, the problem cropped up when only one of these outputs got spent. This entire leveldb record had to be rewritten minus the spent output.

In this newer format, there is only one delete for a specific output index and transaction hash, i.e. just delete one leveldb record, no update required.

Let's take care of a small problem, which is a varying number length. The number, which is of a variable size is written first and then followed by the hash. Presently, we have taken one byte only for the number. This value is stored in the hash0 variable. The rest of the string is the transaction hash.

But what happens when the output index needs two bytes for size. Now comes a simple kludge. The size of the remaining string is checked to be larger than 32 bytes. If yes, then the next byte is picked up and 128 is subtracted from it. The hash size is also reduced by 1, it should be 32 bytes. Then we add one to the second byte length and multiply it by 128. This becomes the offset index.

Now for the value.

This is how the comments in the file coins.h looks like.

```
/**
 * A UTXO entry.
 *
 * Serialized format:
 * - VARINT((coinbase ? 1 : 0) | (height << 1))
 * - the non-spent CTxOut (via CTxOutCompressor)
 */
```

You can ignore the comments and look at a function called `Serialize` in the source code. In the function, the actual height of the block is multiplied by 2. 1 is added to it if it is a Coinbase transaction. The initial bytes are for the block height and our loyal `base128` function decodes. It is not the actual height but a value multiplied by 2, so we now divide by 2.

You can verify the same with the blockchain explorer, this height is of the block enclosing the above transaction.

The processed bytes are printed using the offset variable. Then the `base128a` function reads the Bitcoin value. Once again, it is a total transaction value. Finally, the remaining value minus the first byte is the output script.

Now that we have figured out one transaction output, let's run the next program to validate our new findings acquired from the source code.

```
ch3603.py
import leveldb
import subprocess
import json
def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)
def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n
def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
```

```

while j < len(v):
    finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
    j = j + 1
    k = k + 1
    if k == 8:
        k = 0
    return finalstr
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)
def displaychainstate( finalstr , hash , show):
    finalstr = finalstr.decode('hex')
    if show == 1:
        print "Value %s" % (finalstr.encode('hex'))
        (nHeight , offset) = base128(finalstr, 0)
    if show == 1:
        print "Final %s" % finalstr[offset:].encode('hex')
        (bvalue, offset) = base128a(finalstr, offset)
    if show == 1:
        print "bvalue %d offset %d" % (bvalue, offset)
        nHeight = nHeight / 2
    if nHeight < 0 or nHeight > 1000000:
        print "Error in height %d" % nheight
        exit(0)
    if bvalue < 0:
        print "Error in Bitcoin Value %d" % bvalue
        exit(0)
xorkey = ""
cnt = 0
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstate3")
for k,v in db.RangeIter():
    cnt = cnt + 1
    if k[:1] == 'C':
        finalstr = xorstring(v , xorkey)
        hash = k[1:]
        if cnt % 10000 == 0:
            print "====Checking for UTXO set record number %d" % (cnt)
            displaychainstate(finalstr , hash[:-1] , 0)

```

```
        elif ord(k[:1]) == 14:
            xorkey = v[1:]
#all passed
```

All 51 million leveldb UTXO records are scanned. The block height of the transaction is obtained. We cannot run bitcoin-cli twice to check this height as there is too much data. The height is checked to be within the range of 0 to 1 million. If yes, we believe we have made no mistake in reading the height. We check that the Bitcoin value is a positive number.

When the show variable has a value of 1, it will execute the print command and display the text. This is pertinent as 51 million iterations display lots of data. This program gets over in a jiffy. The next one takes days, maybe because it does very little.

```
ch3604.py
import leveldb
import subprocess
import json
def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            amt = decompressamout(n)
            return (amt, offset)
def decompressamout(x):
    if x == 0:
        return 0
    x = x - 1
    e = x % 10
    x /= 10
    if (e < 9) :
        d = (x % 9) + 1
        x /= 9
        n = x*10 + d
    else:
        n = x+1
    while (e):
        n *= 10
        e = e - 1
    return n
def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
```

```

while j < len(v):
    finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
    j = j + 1
    k = k + 1
    if k == 8:
        k = 0
    return finalstr
def base128(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
        offset = offset + 1
        n = (n << 7) | (chData & 0x7F)
        if chData & 0x80 == 128:
            n = n + 1
        else:
            return (n,offset)
def displaychainstate( finalstr , hash , show):
    global cnt
    finalstr = finalstr.decode('hex')
    hash0 = ord(hash[0])
    hash = hash[1:]
    if len(hash) != 32:
        length = ord(hash[0]) - 0x80
        hash = hash[1:]
        hash0 = hash0 + 128 * (length + 1)
    (nHeight , offset) = base128(finalstr, 0)
    (bvalue, offset) = base128a(finalstr, offset)
    publickey = finalstr[offset:]
    publickey = publickey.encode('hex')
    raw = subprocess.check_output(["bitcoin-cli", "gettxout" , hash.encode('hex'),
    str(hash0)])
    if len(raw) == 0:
        return
    a = json.loads(raw)
    bitcoinvalue = a['value'] * 100000000.0
    bitcoinvalue = round(bitcoinvalue , 2)
    bitcoinvalue = int(bitcoinvalue)
    if bitcoinvalue != bvalue :
        print "=====Mismatch in Bitcoin Values %d:%d cnt %d" % (bvalue ,
        bitcoinvalue , cnt)
    c = ""
    if a['scriptPubKey']['type'] == 'scripthash':
        if len(a['scriptPubKey']['hex']) == 46:
            c = a['scriptPubKey']['hex'][4:-2]
            publickey = publickey[2:]

```

```
elif a['scriptPubKey']['type'] == 'pubkeyhash':
    if len(a['scriptPubKey']['hex']) == 50:
        publickey = publickey[2:]
        c = a['scriptPubKey']['hex'][6:-4]
    elif len(a['scriptPubKey']['hex']) == 52:
        publickey = publickey[2:]
        c = a['scriptPubKey']['hex']
    elif a['scriptPubKey']['type'] == 'pubkey':
        if len(publickey) < len(a['scriptPubKey']['hex']):
            publickey = publickey[2:]
            c = a['scriptPubKey']['hex'][4:68]
        else:
            publickey = publickey[2:]
            c = a['scriptPubKey']['hex']
    elif a['scriptPubKey']['type'] == 'multisig':
        extra = 0
        if ord(publickey.decode('hex')[0]) == 0x80:
            extra = 1
        publickey = publickey[2:]
        length = ord(publickey.decode('hex')[0])
        length = (length - 2) + 128 * extra
        publickey = publickey[2:]
        c = a['scriptPubKey']['hex'][0:length * 2]
    elif a['scriptPubKey']['type'] == 'witness_v0_scripthash':
        publickey = publickey[2:]
        c = a['scriptPubKey']['hex']
    elif a['scriptPubKey']['type'] == 'witness_v0_keyhash':
        publickey = publickey[2:]
        c = a['scriptPubKey']['hex']
    elif a['scriptPubKey']['type'] == 'nonstandard':
        return
    else:
        print "Type is %s %d" % (a['scriptPubKey']['type'], cnt)
        return
    if c == publickey:
        #print "Script Public Key matches"
        pass
    else:
        print "—Script Public Key mismatch at %d" % cnt
        print "Hash Type is %s" % a['scriptPubKey']['type']
        print a['scriptPubKey']
        print "publickey %s" % publickey
        print "c %s" % c
        print a
xorkey = ""
cnt = 0
```



```

db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstate3")
for k,v in db.Rangelter():
    cnt = cnt + 1
    if cnt % 1000000 == 0:
        print "====Checking for UTXO set record number %d" % (cnt)
        if k[:1] == 'C':
            finalstr = xorstring(v , xorkey)
            hash = k[1:]
            displaychainstate(finalstr , hash[:-1] , 0)
        elif ord(k[:1]) == 14:
            xorkey = v[1:]
            =====Checking for UTXO set record number 51320000

```

This program takes about 4 days or more to run. We extract the transaction hash bearing in mind that the output index in this hash may take up 2 bytes. The height and bitcoin values, both are acquired. The `gettxout` method is called with the transaction hash and the index of the output, in a string format. The string value is stored in the `hash0` variable.

This json formatted output has a value field that gives out the Bitcoins associated with this output. A UTXO only deals with Outputs, the inputs indirectly point to outputs.

We wonder why we must round up the floating-point number and then convert it into a int. We have an aversion to floating point numbers. The Bitcoins are stored as Satoshis. The Bitcoin value associated with the value of the key and the Bitcoin value returned by the method `gettxout` are checked.

Like before, the type of every transaction is checked. There have been instances where the outputs not part of the blockchain are still lurking around.

Let's take each type and look at the exceptions also. We start with the `scripthash` type.

```

{u'bestblock':
u'00000000000000000000c49ed52928a7b766487f0cf6bbdb4de0cdf391610ca70d',
u'coinbase': False, u'confirmations': 502, u'value': 0.00381422, u'scriptPubKey':
{u'reqSigs': 1, u'hex': u'a9140baecb9d6a8a36d33e8332cb5f98ad1c86a6b6ee87',
u'addresses': [u'32knke37csk72ZTYJnM1Dv9Qy4assAXy'], u'asm': u'OP_HASH160
0baecb9d6a8a36d33e8332cb5f98ad1c86a6b6ee OP_EQUAL', u'type': u'scripthash'}}
010baecb9d6a8a36d33e8332cb5f98ad1c86a6b6ee

```

We were expecting a few exceptions to pop up, but the type `scripthash` is well behaved. The first byte is the opcode `OP_HASH160` followed by the length, 20 bytes. Then comes the RipeMD hash. The last byte is another `OP_EQUAL`. We ignore the first 4 characters and the last two, in other words, the first 2 and the last byte.

Now comes the type `pubkeyhash`

```

{u'bestblock':
u'00000000000000000000c49ed52928a7b766487f0cf6bbdb4de0cdf391610ca70d',
u'coinbase': False, u'confirmations': 11077, u'value': 0.00065279, u'scriptPubKey':
{u'reqSigs': 1, u'hex': u'76a914cbc2986ff9aed6825920aece14aa6f5382ca558088ac',
u'addresses': [u'1KaPHfvVWNZADup3Yc26SfVdkTDvvHySVX'], u'asm': u'OP_DUP
OP_HASH160 cbc2986ff9aed6825920aece14aa6f5382ca5580 OP_EQUALVERIFY
OP_CHECKSIG', u'type': u'pubkeyhash'}}
00cbc2986ff9aed6825920aece14aa6f5382ca5580

```



```
{u'bestblock':
u'000000000000000008be44bcfd2f0a72decbad2e57126b9d315154a805088e1',
u'coinbase': False, u'confirmations': 165541, u'value': 7.8e-05, u'scriptPubKey':
{u'reqSigs': 1, u'hex':
u'512102b37d95ab362de46450f8e6210b4a562b2d400943f7af188d398dfd9498c83ce12
11c434e5452505254590000000000000d806c1d50000005d653bd9080000000052ae',
u'addresses': [u'1Kp6gQPt1R4j1bG9kafWtUwjuzgnkS6Exi'], u'asm': u'1
02b37d95ab362de46450f8e6210b4a562b2d400943f7af188d398dfd9498c83ce1
1c434e5452505254590000000000000d806c1d50000005d653bd90800000000 2
OP_CHECKMULTISIG', u'type': u'multisig'}}
4d512102b37d95ab362de46450f8e6210b4a562b2d400943f7af188d398dfd9498c83ce1
211c434e5452505254590000000000000d806c1d50000005d653bd9080000000052a
e
```

Now comes the big daddy of output lengths, the multisig type. The problem is that because the length of the hex field can be larger than 128 we may come across a length byte, 0x80. If we meet it, we simply set the extra variable to 1. The current byte is skipped and the next byte is taken as the length of hex field, plus 128 if the variable extra is 1. That many bytes are read from the hex field keeping in mind that it is an encoded string. The key value is read.

```
{u'bestblock':
u'000000000000000000028eabafdae2f2416144ac7c8e930f32ffc94abed1e2848',
u'coinbase': True, u'confirmations': 314102, u'value': 0.00592608, u'scriptPubKey':
{u'type': u'nonstandard', u'hex': u'736372697074', u'asm': u'OP_IFDUP OP_IF
OP_2SWAP OP_VERIFY OP_2OVER OP_DEPTH'}}
0c736372697074
```

We have no choice but to skip all nonstandard outputs, there are many of them.

Finally, we see two new hash types on the gallery.

```
{u'bestblock': u'00000000000000000009459ee966fd32b51633da3217ac4e33d165a6a97afb81',
u'coinbase': False, u'confirmations': 2479, u'value': 0.00021352, u'scriptPubKey':
{u'type': u'witness_v0_scripthash', u'hex':
u'00209147028dba70cfae690f9c6b9bc977e354ee06c2beabdaf28c540c5f22803595', u'asm': u'0
9147028dba70cfae690f9c6b9bc977e354ee06c2beabdaf28c540c5f22803595'}}
2800209147028dba70cfae690f9c6b9bc977e354ee06c2beabdaf28c540c5f22803595
```

The first one is called witness_v0_scripthash. Nothing can get easier than this, the first byte of the keys value is skipped and the hex field is compared as is.

```
{u'bestblock':
u'00000000000000000001cfdd88dba00a23f51b89c9a396da5c84b1d0ad5e4ed23',
u'coinbase': False, u'confirmations': 4352, u'value': 0.00435565, u'scriptPubKey':
{u'type': u'witness_v0_keyhash', u'hex':
u'001425c14e09f4a40a8d03e91130f6aa98eea6bab863', u'asm': u'0
25c14e09f4a40a8d03e91130f6aa98eea6bab863'}}
1c001425c14e09f4a40a8d03e91130f6aa98eea6bab863
```

The last kid on the block is `witness_v0_keyhash`. Here also no major surgery. One observation made here is that the newer types stored in the output and the keys value are sort of the same, not like the older types. What got our goat was getting one output per output type. This took a long time.

Back to the drawing board. The release notes give very clear information where they describe the older UTXO model as per transaction and the newer one as per model.

Our take on it. Let's suppose that we have a transaction with 10 outputs, 7 of which are unspent. In the older model, there would be only one record in the leveldb database. Apropos to that, the newer model has 7 leveldb records. Our understanding was that, it would increase the space usage exponentially. But such is not the case because the newer versions of leveldb handles cases very efficiently where the prefix of the key changes (the output index) but the rest of the key (the transaction hash), remains the same. The credit is given to the encoding schemes used in leveldb.

The older UTXO set was first introduced in Bitcoin Core version 0.8 and released on the 19th of Feb 2013. Changing the UTXO set is not for the fainthearted. We have mentioned this n number of times in the past, the blockchain stored over 200 GB on disk is a waste of disk space. The UTXO set is the blockchain as it has information on the quantum of money each Bitcoin address has control over.

If ever there is a bug in the UTXO software, the unspent coins get lost only on your machine but for someone running an older UTXO version, your coins are still present. This can split the network. There is no space for any bugs in the software. There were over 2 months of review, 145 comments and counting. The Core developers used a testing method called mutation testing. They scanned the UTXO code line by line and introduced a bug. Then they made sure that their own tests caught the bug. The question is, you write tests to test the software, but who is testing the test.

It makes sense to use the latest version of leveldb which is 1.2, released on the 2nd of March 2017. Nearly all internal databases use leveldb. This release like SHA hashes also uses the SSE 4.2 instruction set.

Never be the first person to try out new software. When the new UTXO set was first introduced, the size of the chainstate folder was not 2.8GB but 5.8GB. The leveldb was not aggressive in removing the older keys. A hidden command `forcecompactdb` (more on hidden commands later) had to be executed forcing the delete of the keys in leveldb. However, if you are running version 0.14 or earlier, it takes about 2-3 minutes to upgrade the chainstate folder. But, if you want to go back to the older UTXO set, you must reindex the chainstate and even god will not help you there.

Now for some numbers. There is a 30-40% increase in performance over the older model. It also consumes 10-20% less memory and it flushes to disk less often. The older model tried to save on key space by reusing values like the transaction hash and the height and the Coinbase. The chainstate folder size increased by 15%. Whenever you flush any memory to disk, there is a spike in memory use. The total cache memory also reduces by half as we have to save to disk. The newer model reduces the amount of disk flushing by half. A simpler model can predict memory usage also. The final advantage of a simpler model is less CPU overhead when dealing with unspent outputs which are spent. All in all, a good job done.

Non-Atomic Flushing

```
ch3605.py
import leveldb
def reversehash(hash):
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))

def base128a(ans , offset):
    n = 0
    while True:
        chData = ord(ans[offset:offset + 1])
```

```

    offset = offset + 1
    n = (n << 7) | (chData & 0x7F)
    if chData & 0x80 == 128:
        n = n + 1
    else:
        amt = decompressamt(n)
        return (amt, offset)

def xorstring(v , xorkey):
    j = 0
    k = 0
    finalstr = ''
    while j < len(v):
        finalstr = finalstr + "%02x" % (ord(v[j]) ^ ord(xorkey[k]))
        j = j + 1
        k = k + 1
        if k == 8:
            k = 0
    return finalstr

def displaychainstate( finalstr ):
    print "%d" % int(finalstr[:2])
    finalstr = finalstr[2:]
    final1 = finalstr[0:64]
    final2 = finalstr[64:]
    print reversehash(final1)
    print reversehash(final2)

xorkey = ''
db = leveldb.LevelDB("/Users/vijaymukhi/Desktop/chainstate")
for k,v in db.RangeIter():
    if k[:1] == 'H' :
        print k , len(k)
        finalstr = xorstring(v , xorkey)
        displaychainstate(finalstr)
    elif ord(k[:1]) == 14:
        xorkey = v[1:]

```

Output

```

H 1
2
0000000000000f2ef239bbb89b5bda69361854e5e1e240c995b62b610fa2913c
000000000000037fe63c72733aad12a0c03783fe0c6ea36d5b894ca0ad3d886c

```

This program is simply a repetition of the code displaying the UTXO set. There is one small twist in the tale. A new key has been introduced called the head blocks key or H. This key name is stored in the file txdb.cpp. The key is created in the source code and it is initialized in the following way.

```
static const char DB_HEAD_BLOCKS = 'H';
```

Fortunately for us, this H key is of size 1 only. There is no key data after this key. We hope all keys would be of length 1. Then follows the 65 bytes of data in the value associated with this key. The first byte being 2 indicates that there are two objects or in this case, two hashes following. A hash will always be 32 bytes large and hence the value of the H key gives a starting block hash and an ending block hash. We don't have to tell you anymore that we must reverse this hash.

The first hash starting with f2 is the hash of block 131299 and the hash starting with 37 is of block 131294. At startup when bitcoind sees an H key, it knows that it must replay blocks from 131294 to 131299.

The bitcoind showed the following at another point in time:

```
2017-09-28 07:30:26 Replaying blocks
2017-09-28 07:30:26 Rolling forward
000000003b440de466fbd31a7889c87aabeadd342b418d2986127ead95bfb859 (41794)
2017-09-28 07:30:26 Rolling forward
000000006227ca03272b8b0e2997581f7eb7591ee30aa8af42d5c8a7b531a963 (41795)
```

To figure out this key, we looked at a function called BatchWrite in txdb.cpp. This function is called whenever bitcoind wants to write multiple leveldb records. So, in this function there is a line:

```
batch.Write(DB_HEAD_BLOCKS, std::vector<uint256>{hashBlock, old_tip});
```

On execution, this line creates a key called H and writes the ending and starting block hash values.

```
batch.Erase(DB_HEAD_BLOCKS);
```

At the end of this function, the H key gets deleted. We tried exiting out but the key H always got deleted. So, we decided to create a key called V in the same way and hoped that nobody would delete it.

The following code tries to accomplish this magic. C++ makes you stand on our heads to get stuff done.

```
batch.Write(DB_HEAD_BLOCKS, std::vector<uint256>{hashBlock, old_tip});
printf("+++++++txdb.cpp BatchWrite Start %s %s\n",
hashBlock.ToString().c_str(), old_tip.ToString().c_str());
cnt++;
if (cnt == 3) {
    printf("=====Now is the time\n");
    batch.Write('V', std::vector<uint256>{hashBlock, old_tip});
}
```

The only way to understand Bitcoin Core is by getting your feet wet in the source code. A global variable called cnt is created and after 3 accesses, we write out a key called V and decipher it. The printf with the plus signs is our marker indicating our code is getting called. Having lots of bitcoind output is a bad idea.

This flushing of the UTXO cache on disk is intermittent. So, we start with zero Bitcoin blocks. The function BatchWrite gets called the first time when the height is 233462 and the next time, at block 233459, pretty close.

The randomness is at the second function call. The first time it gets called at block height 274674 but the third call of the BatchWrite function was at block height 297079. No one can predict when this function will get called.

Now for a theory class.

According to the core grapevine, the new UTXO set preceded the H key.

What role does caching play in Bitcoin? Let's take a transaction that has two unspent outputs, which get spent in the

next block. If these two unspent outputs are immediately written into the chainstate folder we are writing to disk. But if these unspent outputs are kept in memory or in a buffer, they may get spent in the next 10 minutes. In such a situation, we are saving on a disk write as these two spent outputs now do not belong to the UTXO set.

The performance comes in because there are less disk writes as the writes are delayed. Try out this experiment by downloading the Bitcoin block chain from scratch. You will realize that for the first 200,000 blocks, the size of the chainstate folder is effectively 0. There are no writes of the UTXO set.

We need to keep in mind that while writing a buffer or cache to disk, the power can go off at any point in time. A recovery from this inconsistent state is a must as some part of the UTXO set is not written. Ideally, the writes should be in ranges and there should be a flag which marks successful writes to the UTXO set, then only a certain block numbers/range would be rewritten. You can therefore recover from a certain block height from where the UTXO set has been written to disk correctly. However, you would want the block height to be the latest if possible, but what is most important is that you must have a block height where the state is good.

In Bitcoin Core, when the cache gets full, there is a force write of the entire UTXO set to disk in one go (with a prayer). This writing to disk takes about 100 milliseconds these days. We assume that nothing went wrong but things will go wrong. The way out is to create a database transaction which would write the data to disk.

A database transaction comes with own set of conditions. It needs to first save the entire UTXO set in memory. We have two copies now. So, if your cache size is 100 MB (the docs use the 1GB example), then we cannot use the full 100 MB for the cache, only 50 MB allowed, the other 50 MB is used only during flushing. We are wasting half of our cache all the time doing nothing.

```
2017-09-27 11:10:49 UpdateTip: new
best=00000000000000aa06b46226fa42cb71e1e83ed07a59d75d46b95398efb676fc
height=233476
txdb.cpp BatchWrite Start
00000000000000aa06b46226fa42cb71e1e83ed07a59d75d46b95398efb676fc
0000000000000000000000000000000000000000000000000000000000000000

2017-09-27 11:33:10 UpdateTip: new
best=0000000000000003bdd925491f64b6d3e9fb63a1f86c96708561e7d24c18b5f8
height=274674
txdb.cpp BatchWrite Start
0000000000000003bdd925491f64b6d3e9fb63a1f86c96708561e7d24c18b5f8
00000000000000aa06b46226fa42cb71e1e83ed07a59d75d46b95398efb676fc
```

Let's understand what happens behind the scenes with some real data. The bitcoind server reveals that the first write to disk was done at block height 233476. The Update Tip shows the last block height downloaded on our disk.

The H key marks the hash of this block as the ending hash value; the starting block hash is all 0's which is the start of the blockchain. In case of an unpleasant event like power failure happening at this moment, the H key would announce that the UTXO set has no data as data from block 0 to block 274674 is still not written. There is nothing to roll or rollback in any direction.

At the second call of the BatchWrite function, the second hash belongs to block number 233476. The highest or the current active tip or block is number 274674. The key H discloses that block 233476 was written to disk correctly. There was a problem while writing data from block number 233476 to block number 274674. So, roll back or roll forward or roll sideways, your call. There is a small database that has a continuous record of information on the last block successfully saved.

The Bitcoin Core developers realized after a long time though, that the blockchain is a write ahead log. The order of UTXO records stored in the chainstate folder is not important. When the writing is in process, the UTXO set keeps track of the starting block number. The last downloaded block number is also saved.

At power outage, we simply add the changes that never got written to disk in the first place by reading the block data from disk and re-apply the changes. The UTXO set contains those transactions whose output is unspent. When the output is spent, the record is deleted. The simplicity of the newer UTXO set makes the H key useful. We do not have to change or update a record.

Two identical copies of a record or adding a record twice does not affect the UTXO set. The future is very bright as now we can flush the UTXO set at every block. The Core designers have leveldb in their cross hairs, there is nothing that does the job better than leveldb today, but tomorrow is another day. Bitcoin is not wedded to leveldb.

To sum up, there are no atomic writes with the UTXO set. Before starting the caching process, a H key identifies the UTXO's blocks, which are to be flushed to disk. This UTXO cache is written in batches of 16MB and in the end, the key is deleted. The presence of this key spells trouble and we simply rollback or roll forward the whole process. Recovery is impossible without this H key.

The release notes on this non-atomic flushing didn't give much. So, we read on the pull requests notes, another way of saying committing code. The pull request is identified as 10148 and the URL is <https://github.com/bitcoin/bitcoin/pull/10148>. We understood when we saw things with our own eyes, the H key values. Seeing is believing.

We created the V key after reading a comment by sipa where he actually placed an `exit(0)` in the code and bitcoind recovered successfully. We like reading that sipa added new arguments, don't have to take permission from any gods.

You can't hold a candle to the Bitcoin Core developers if you don't read the Pull Requests. This is where they literally explain the code line by line. These PRs make for very difficult reading.

Let's ban the enemy.

In the two chapters on Bitcoin networking, we learnt that one peer connects to another by initially exchanging a version packet. There is a field called services in this version packet structure. The value of this field was always 1 and hence we ignored it.

Let's suppose that we do not want to connect to some undesirable peers till the 1st of August 2018, the time is represented by a number in seconds of 1533096000.

In segwit2x code base we see the following line.

```
./src/protocol.h:271:  // NODE_SEGWIT2X supports segwit2x
./src/protocol.h:272:  NODE_SEGWIT2X = (1 << 7),
```

This is one more way of creating a variable and setting the 7th or 8th bit on. The actual value of the variable is 0x0080. You decide which way is more readable. The << is the bitwise left shift operator.

When a segwit2 peer sends a version packet across, it ensures that the 7th bit (counting from 0) is set to 1. This will inform the node or peer on the other side that the peer support segwit2x. Bad idea as you will soon see.

Now in the Bitcoin Cash or Bitcoin ABC source code, you will see the following:

```
// NODE_BITCOIN_CASH means the node supports Bitcoin Cash and the
// associated consensus rule changes.
// This service bit is intended to be used prior until some time after the
// UAHF activation when the Bitcoin Cash network has adequately separated.
// TODO: remove (free up) the NODE_BITCOIN_CASH service bit once no longer
```



```
// needed.
./src/protocol.h:276:  NODE_BITCOIN_CASH = (1 << 5),
```

The services field in every Bitcoin Cash node has the 5th or 6th bit on to identify itself. If one does not want to connect to any node that is segwit2x or Bitcoin Cash compliant, simply check the services field in the version packet. If bit 5 or bit 7 is on then you are sleeping with the enemy and you can refuse the connection. This code is visible in the file `net_processing.cpp` line number 1263 in Bitcoin Core.

```
if (nServices & ((1 << 7) | (1 << 5))) {
    if (GetTime() < 1533096000) {
        // Immediately disconnect peers that use service bits 6 or 8 until August 1st, 2018
        // These bits have been used as a flag to indicate that a node is running incompatible
        // consensus rules instead of changing the network magic, so we're stuck disconnecting
        // based on these service bits, at least for a while.
        pfrom->fDisconnect = true;
        return false;
    }
}
```

The counting in the comments start from 1 and not 0. A different network magic number would be most suitable here for segwit2x and Bitcoin Cash but no one is listening. The Bitcoin core coders prefer using words like incompatible consensus rules. Their reasoning is that most of the nodes run incompatible software, so all this is waste of time.

Most people believe that Bitcoin Cash is a hard fork that is cast in stone. There are chances that Bitcoin Core implementation will receive an 8MB block and then get rejected. However, segwit2x hard fork will be activated in November 2017 hopefully. Some people think that the segwit2x hard fork will never happen.

One of `printf` statements showed the following in the output of `bitcoind`.

```
2017-09-21 10:45:55 receive version message: /Satoshi:0.14.1/: version 70015,
blocks=486291, us=60.243.109.55:63215, peer=8
+++++Sleeping with the enemy
2017-09-21 10:46:04 ProcessMessages(version, 106 bytes) FAILED peer=7
```

There are Bitcoin Cash or segwit2x nodes that want to send blocks to us but we reject it. But if we comment out the last two lines in the if statement, things will change.

Fee Estimates

Our work on the file `fee_estimates.dat` has gone down the drain as the file format has changed completely. In short, the already difficult code to estimate Bitcoin fees has been rewritten and is now made more intelligent. It takes more data into account while computing a fee. We get better fee estimates now. We could not understand the earlier code, so we doubt if we will ever understand the newer smarter code.

There are two different types of estimates, economical and conservative. The new `estimatesmartfee` method gives the following output.

```
bitcoin-cli estimatesmartfee 2 ECONOMICAL
{
    "feerate": 0.00136692,
    "blocks": 2
}
```

```
bitcoin-cli estimatesmartfee 2 CONSERVATIVE
{
  "feerate": 0.00136692,
  "blocks": 2
}
```

The fees are the same.

```
bitcoin-cli estimatesmartfee 1008 ECONOMICAL
{
  "feerate": 0.00001008,
  "blocks": 221
}
```

We can specify a block target value up to 1008 blocks, but the blocks field stays at 221, the block number where the estimate occurs. That's why the blocks field has a value of 221.

Faster SHA-256 hashes

The Bitcoin people are always seen calculating SHA-256 hashes. What if these hash calculations are speeded up. In the year 2006, Intel added some 54 new instructions called SSE4 or Streaming SMD Extension 4 or SSE4. The acronym SIMD stands for Single Instruction Multiple Data. The bottom line, software runs faster when you use these instructions.

Greg Maxwell admits that they should have had a version of SHA-256 written using SSE4. They finally added this code in a file, sha256_sse4 in the crypto folder. All the statistics and numbers quoted are from the Bitcoin Core team, we have not confirmed any of these numbers as we are not qualified to do so.

The initial block download is faster by 5% and when a new block is connected to the tip, a further saving of 10%. This feature is not activated by default even though the code is present in the code base. The release notes or talks by the Bitcoin Core greets like Greg Maxwell and Pieter Wuille comment on it.

The code was crashing on the Mac OSX. The developers spent over 3 days trying to find out why only on the Mac. They then realized that the C++ compiler was optimizing away the assembler code. The fault was in the capitalization of a label. They plan to test the code on all obscure OS's and then only enable it by default.

While compiling the source code, the flag `—enable-experimental-asm` must be added. According to Dr. Pieter Wuille or sipa, he gets a 50% speedup.

AMD has an actual instruction that computes the SHA hash in hardware and on some chips, Intel also adds this new instruction. The Bitcoin Core plans using this instruction.

We are not here to take sides in the three-way battle between Bitcoin Core or Bitcoin Cash or Segwit2x. We read the tea leaves as we see them. The other two yet do SHA hash computations the old way, even after 11 years some things do not change.

You will see this line in the output generated by bitcoind.

```
2017-09-21 08:16:44 Using the 'sse4' SHA256 implementation
```

Who is blind?

This is nothing to do with Script Validation Caching nor is it a new feature of Bitcoin Core version 0.15. This has more to do with Dr. Pieter Wuille being blind. Yes, you read it right, blind.

This is a piece of code from the file `validation.cpp`, line number 1223, your mileage will vary slightly.

```
size_t nMaxCacheSize = std::min(std::max((int64_t)0, gArgs.GetArg(
    "-maxsigcachesize", DEFAULT_MAX_SIG_CACHE_SIZE) / 2),
    MAX_MAX_SIG_CACHE_SIZE) * ((size_t) 1 << 20);
```

At URL <https://github.com/bitcoin/bitcoin/pull/10192>, you will see the following conversation.

sipa on April 12th

I think the division should be outside of `GetArg`. Otherwise, if you specify `-maxsigcachesize=32`, you end up with a total of 64MiB worth of caches.

gmaxwell also on April 12th

Good, because the division is outside of the `GetArg`. :)

sipa on April 13th

It seems I am blind.

Only geniuses can make fun of themselves. We like nuggets like these.

The top five contributors to version 0.15 are John Newberry (Chaincode 14%), Pieter Wuille (Blockstream 9%), Matt Corallo (Chaincode 8%), Alex Morcos (Chaincode 7%) and Wladimir van der Laan (DCI 7%). It catches attention when you walk into a party and mention their names.

Also keep a watchful eye on Chaincode, they are sparing their key people to work on Bitcoin.

The Block Size War

The whole ugly civil war between the big three, Core Bitcoin, Bitcoin Cash and Segwit2x is all about block size. Let's understand block size by looking at source code of different Bitcoin versions as well as the dark horses. Let's start with Bitcoin Core Version 0.12.

The file `consensus.h` has the following lines.

```
/** The maximum allowed size for a serialized block, in bytes (network rule) */
static const unsigned int MAX_BLOCK_SIZE = 1000000;
```

This variable `MAX_BLOCK_SIZE` is 1 MB and the rule to be enforced is part of the network. In the file called `main.cpp` and in function `CheckBlock`, we come across this error check.

```
if (block.vtx.empty() || block.vtx.size() > MAX_BLOCK_SIZE ||
    ::GetSerializeSize(block, SER_NETWORK, PROTOCOL_VERSION) > MAX_BLOCK_SIZE)
    return state.DoS(100, error("CheckBlock(): size limits failed"),
        REJECT_INVALID, "bad-blk-length");
```

The variable `block` represents the current Bitcoin block. The first check is to make sure it is not empty. The next condition is that the two block sizes should not be larger than 1MB.

Then in the file `miner.cpp` we have one more error check.

```
// Limit to between 1K and MAX_BLOCK_SIZE-1K for sanity:
nBlockMaxSize = std::max((unsigned int)1000, std::min((unsigned
int)(MAX_BLOCK_SIZE-1000), nBlockMaxSize));
```

As a miner, it simply ensures that the block size is 100k less than 1 MB. Any change to the block size must be made here.

In Bitcoin Cash, we assumed that they would change the 1MB to 8MB in the file consensus.h, that was not to be.

```
static const uint64_t ONE_MEGABYTE = 1000000;
/** The maximum allowed size for a transaction, in bytes */
static const uint64_t MAX_TX_SIZE = ONE_MEGABYTE;
/** The maximum allowed size for a block, before the UAHF */
static const uint64_t LEGACY_MAX_BLOCK_SIZE = ONE_MEGABYTE;
/** Default setting for maximum allowed size for a block, in bytes */
static const uint64_t DEFAULT_MAX_BLOCK_SIZE = 8 * ONE_MEGABYTE;
```

Finally, the introduction of the new variable called DEFAULT_MAX_BLOCK_SIZE whose value is 8MB. The function CheckBlock in the validation.cpp file has this code.

```
auto currentBlockSize =
    ::GetSerializeSize(block, SER_NETWORK, PROTOCOL_VERSION);
if (currentBlockSize > nMaxBlockSize) {
    return state.DoS(100, false, REJECT_INVALID, "bad-blk-length", false,
        "size limits failed");
}
```

The variable nMaxBlockSize is initialized to the variable DEFAULT_MAX_BLOCK_SIZE or 8MB. The function GetSerializeSize returns the current size of the block. Bitcoin Cash does not allow you to set the block size lower than 1MB. This is from config.cpp, function SetMaxBlockSize.

```
// Do not allow maxBlockSize to be set below historic 1MB limit
// It cannot be equal either because of the "must be big" UAHF rule.
if (maxBlockSize <= LEGACY_MAX_BLOCK_SIZE) {
    return false;
}
nMaxBlockSize = maxBlockSize;
```

Now let's see how the newer Segwit2x handles block sizes. Same file, consensus.h.

```
/** The maximum allowed size for a serialized block, in bytes (only for buffer size
limits) */
static const unsigned int MAX_BLOCK_SERIALIZED_SIZE = (8 * 1000 * 1000);
```

Very simply, it cannot be larger than 8MB. Now we are back to the CheckBlock function in the file validation.cpp.

```
if (block.vtx.empty() || block.vtx.size() > MAX_BLOCK_VTX ||
    ::GetSerializeSize(block, SER_NETWORK, PROTOCOL_VERSION |
    SERIALIZE_TRANSACTION_NO_WITNESS) > MAX_BLOCK_SERIALIZED_SIZE)
    return state.DoS(100, false, REJECT_INVALID, "bad-blk-length", false, "size limits
failed");
```

Three different sizes are checked here. The last check is on the size of a block without witnesses. Is it less than 8 MB? There are two transaction formats, one with witness data and one without witness data.

Finally, the file consensus.h in Bitcoin Core 0.15.

```
/** The maximum allowed size for a serialized block, in bytes (only for buffer size limits) */
static const unsigned int MAX_BLOCK_SERIALIZED_SIZE = 4000000;
```

```
/** The maximum allowed weight for a block, see BIP 141 (network rule) */
static const unsigned int MAX_BLOCK_WEIGHT = 4000000;
```

We have a size that is restricted to 4MB plus a block weight which is also 4MB. The CheckBlock function in file validation.cpp now reads.

```
// Size limits
if (block.vtx.empty() || block.vtx.size() * WITNESS_SCALE_FACTOR >
    MAX_BLOCK_WEIGHT || ::GetSerializeSize(block, SER_NETWORK,
    PROTOCOL_VERSION | SERIALIZE_TRANSACTION_NO_WITNESS) *
    WITNESS_SCALE_FACTOR > MAX_BLOCK_WEIGHT)
    return state.DoS(100, false, REJECT_INVALID, "bad-blk-length", false, "size limits
    failed");
```

The first check is to ensure that the block is not empty. The last check stresses that the size of the block plus transactions without witness data multiplied by 4 must not be larger than 4MB. We are increasing the size of the block as we are not counting witness data in the 1MB limit.

The vtx object is of type CTransactionRef or CTransaction which is a transaction that has no witness data. We check that this transaction size (without witnesses) multiplied by 4 is less than 4MB. Once again, an increase in size of the block.

```
select * from size1 order by size desc limit 5;
no    | size    |      ctime
-----+-----+-----
484383 | 1372967 | Sun Sep 10 01:15:07 2017
484379 | 1314886 | Sun Sep 10 01:04:29 2017
485159 | 1227087 | Thu Sep 14 19:47:13 2017
486549 | 1138227 | Sat Sep 23 06:57:20 2017
485885 | 1097047 | Tue Sep 19 06:23:27 2017
(5 rows)
```

These are the sizes of the five largest blocks obtained by using the same code to find the Bitcoin Cash blocks. All of them are more than 1MB in size thus proving that blocks in Bitcoin Core are now larger than 1MB but nowhere near the 4MB block size.

Wallets and more Wallets

Bitcoin version 0.15 allows you to deal with multiple wallets at the same time. As a revision please run

```
bitcoin-cli listwallets
[
    "wallet.dat"
]
```

The output confirms that the default wallet is wallet.dat.

```
bitcoin-cli getbalance
22.75050670
```

This balance is on the testnet network so it has no value. So far so good. Now run bitcoind as

```
bitcoind -printtoconsole -testnet -wallet=vijay.dat
```

The option wallet allows us to specify the wallet we want to use. In our case, the file vijay.dat does not exist. No problem, it gets created from scratch.

```
bitcoin-cli listwallets
[
  "vijay.dat"
]
bitcoin-cli getbalance
0.00000000
```

This only confirms that we can have different wallets with Bitcoin. So, one wallet for work, one for business. These wallets are separate and in a transaction, nobody knows which wallet is supplying the coins.

But we can go one better. Run bitcoind as

```
bitcoind -printtoconsole -testnet -wallet=vijay.dat -wallet=wallet.dat
```

We have now specified two different wallets. This is what the output of bitcoind shows us.

```
2017-09-21 11:16:52 Using wallet vijay.dat
2017-09-21 11:16:52 Using wallet wallet.dat
bitcoin-cli listwallets
[
  "vijay.dat",
  "wallet.dat"
]
```

The method listwallets confirms that there are two wallets active at the same time.

```
bitcoin-cli getbalance
error code: -19
error message:
Wallet file not specified (must request wallet RPC through /wallet/<filename> uri-path).
Try adding "-rpcwallet=<filename>" option to bitcoin-cli command line.
```

An error pops up as we did not specify the wallet with the getbalance method.

```
bitcoin-cli -rpcwallet=wallet.dat getbalance
22.75050670
```

The rpcwallet option is used and the balance is shown. The next best thing is to place this option, rpcwallet in bitcoin.conf. We add this line to bitcoin.conf

```
rpcwallet=wallet.dat
```

Now no error when a wallet is not given.

```
bitcoin-cli getbalance
22.75050670
```

In Bitcoin Version 0.12, we could change our mind about the miner fee for confirmation after sending the transaction.

This was allowed as the sent transaction was to be confirmed yesterday. Fee increase or decrease was associated with a transaction.

In Version 0.14, there was a RPC method introduced called bumpfee. Finally, in Version 0.15 they made sure the Bitcoin GUI or bitcoin Qt also allowed us to change or replace the fee. The Bitcoin Core Developers like us prefer using command line utilities and hence the Bitcoin GUI is treated as a step child.

When an option is marked for advanced users, it means it is only available through bitcoind and bitcoin-cli. We go a step further, we build our own version of bumpfee.

```
bitcoind -printtoconsole -testnet -walletrbf
```

We run bitcoind with the option testnet as it gets very expensive to use real Bitcoins today. We use the walletrbf option to enable Replace-By-Fee.

```
bitcoin-cli -testnet sendfrom "" mv9ccDUpNdVoU8i6wU18g1yQjAXZ2R6cAU 0.5
```

Then, the sendfrom method sends Bitcoins from the default account to any Bitcoin address. We get the following transaction hash.

```
fead74b0fdc1021edb671203b5f46313687ab89931045e786bc22116290b2f56
```

Immediately, we run the next command

```
Bitcoin-cli -testnet bumpfee fead74b0fdc1021edb671203b5f46313687ab89931045e786bc22116290b2f56
{
  "txid": "c8c833a7c9257910f78c45100951542a31ed3681e51f3cd0280f8b0651b8543d",
  "origfee": 0.00000374,
  "fee": 0.00002247,
  "errors": [
  ]
}
```

The method bumpfee is passed the received transaction hash. Another transaction hash beginning with c8c8 is returned. You can also see that the transaction fee is bumped from .00000374 to 0.00002247. Let's check these two transactions in a testnet browser.

We are told in stark red color that transaction hash fead74 is dangerous, it is a double spend, so accept at your own risk. What's more, we also told that transaction hash c8c8 is also a problem. But this transaction hash has multiple confirmations.

This gets clearer when we run

```
bitcoin-cli -testnet gettransaction fead74b0fdc1021edb671203b5f46313687ab89931045e786bc22116290b2f56
```

```
{
  "amount": 0.00000000,
  "fee": -0.00000374,
  "confirmations": -455,
  "trusted": false,
  "walletconflicts": [
    "c8c833a7c9257910f78c45100951542a31ed3681e51f3cd0280f8b0651b8543d"
  ],
}
```

```
    "bip125-replaceable": "yes",
    "replaced_by_txid": "c8c833a7c9257910f78c45100951542a31ed3681e51f3cd0280f8b0651b8543d",
    "details": [
      {
        "account": "",
        "address": "mv9ccDUpNdVoU8i6wU18g1yQjAXZ2R6cAU",
        "category": "send",
        "amount": -0.50000000,
        "label": "",
        "vout": 1,
        "fee": -0.00000374,
        "abandoned": false
      }
    ],
```

This makes it very clear that the blockchain stores lots and lots of redundant information about our transaction. The amount is 0, it is bip125 replaceable and conflicts with another transaction hash.

When we run the next command on the new transaction hash

```
bitcoin-cli -testnet gettransaction
c8c833a7c9257910f78c45100951542a31ed3681e51f3cd0280f8b0651b8543d
{
  "amount": 0.00000000,
  "fee": -0.00002247,
  "confirmations": 465,
  "blockhash": "00000000000101a223e5356cbdfb3aa6ab4f4ded747c22950cfe3daba34eac16",
  "blockindex": 8,
  "blocktime": 1505982456,
  "txid": "c8c833a7c9257910f78c45100951542a31ed3681e51f3cd0280f8b0651b8543d",
  "walletconflicts": [
    "fead74b0fdc1021edb671203b5f46313687ab89931045e786bc22116290b2f56"
  ],
  "time": 1505982444,
  "timereceived": 1505982444,
  "bip125-replaceable": "no",
  "replaces_txid": "fead74b0fdc1021edb671203b5f46313687ab89931045e786bc22116290b2f56",
  "details": [
    {
      "account": "",
      "address": "mv9ccDUpNdVoU8i6wU18g1yQjAXZ2R6cAU",
      "category": "send",
      "amount": -0.50000000,
      "label": "",
      "vout": 1,
      "fee": -0.00002247,
      "abandoned": false
    }
  ],
```



```
{
  "account": "",
  "address": "mv9ccDUpNdVoU8i6wU18g1yQjAXZ2R6cAU",
  "category": "receive",
  "amount": 0.50000000,
  "label": "",
  "vout": 1
},
```

We get one more Bitcoin address.

```
bitcoin-cli -testnet getrawtransaction
fead74b0fdc1021edb671203b5f46313687ab89931045e786bc22116290b2f56 1
error code: -5
error message:
No such mempool or blockchain transaction. Use gettransaction for wallet
transactions
```

The getrawtransaction method with the transaction hash starting with fead gives an error. But the same transaction with hash c8c8 has no problem and returns the same standard output.

As an example, please create a normal transaction using the sendfrom method and then run the gettransaction method on the transaction hash. We get this output.

```
"walletconflicts": [
],
"bip125-replaceable": "no",
// The mining code doesn't (currently) take children into
// account (CPFP) so we only consider the fees of
```

Like RBF, there can be an option where the Child transaction Pays For the Parent transaction. The comments in the file validation.cpp state that mining code currently does not support CPFP.

What's in a name

When we run bitcoind as

```
bitcoind -printtoconsole -worldsucks
```

There is no option called worldsucks and bitcoind does not complain. It also gives no error. Been there, done that.

```
bitcoind -h
```

When we run the above command, we see lots and lots of help.

```
-vijaymukhi
Vijay Mukhi is a embecile
```

The last two lines add an option called vijaymukhi with some text. We added an extra option by simply adding just one to the file init.cpp.

```
strUsage += HelpMessageOpt("-rpcthreads=<n>", sprintf(_("Set the number of
threads to service RPC calls (default: %d)"), DEFAULT_HTTP_THREADS));
strUsage += HelpMessageOpt("-vijaymukhi", sprintf(_("Vijay Mukhi is a embecile")));
```

It is quite simple to add an option to the bitcoind program. There is a better and more scientific way of doing the same but it is old hat. Bitcoind also has a lot of undocumented options that do not show up in the help. These are found in the contrib folder and check-doc.py file.

```
# list unsupported, deprecated and duplicate args as they need no documentation
SET_DOC_OPTIONAL = set(['-rpcssl', '-benchmark', '-h', '-help', '-socks', '-tor',
'-debugnet', '-whitelistalwaysrelay', '-prematurewitness', '-walletprematurewitness',
'-promiscuousmempoolflags', '-blockminsize', '-dbcrashratio', '-forcecompactdb',
'-usehd', '-sonalmukhi'])
```

Bitcoind responds to these options but they are not visible in the help and hence there is no documentation for these options. The option sonalmukhi has no effect at all.

The option forcecompactdb is an undocumented option not present in the help. But if we run bitcoind as

```
bitcoind -printtoconsole -forcecompactdb
```

The bitcoind command shows the following output and then there is a very long wait.

```
2017-09-21 12:09:50 Starting database compaction of /Volumes/VijaySamsun/Bitcoin/blocks/index
```

The end of openssl

Our heart bleeds for openssl. The Heartbleed bug was one of the biggest bugs ever found in software. Yes, it was a bug found in openssl. Gradually, Bitcoin moved on and said no more openssl. Its finally reaching the end of the road.

Arguments with a Name

When we run the command without the right parameters, the output gives help on the syntax of the argument.

```
bitcoin-cli getblockhash
Arguments:
1. height      (numeric, required) The height index
```

We have one named argument called height in the help text. It is advisable to use named parameters instead of positional parameters.

```
bitcoin-cli -named getblockhash height=1
00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048
```

The name of the argument is specified with its value. We prefer parameters by position. We will not use the above approach but the release notes talk about it, so shall we. You should also know that not all methods accept named parameters.

```
bitcoin-clibitcoin-cli estimatesmartfee
Arguments:
1. conf_target  (numeric) Confirmation target in blocks (1 - 1008)
```

The method estimatesmartfee has one named parameter called conf_target.

```
bitcoin-cli estimatesmartfee conf_target=3
```

```
error: Error parsing JSON:conf_target=3
```

We get an error as the named parameter is not recognized by the json parser. In the rpc folder, there is a file called client.cpp which contains the following line of code.

```
{ "getblockhash", 0, "height" },
```

That's why the method getblockhash has a named parameter called height. If the method has more than one named parameter, say two, then one more entry gets added, changing the 0 to a 1. For some reason, this is what we see.

```
{ "estimatesmartfee", 0, "nblocks" },
```

The parameter is called nblocks.

```
bitcoin-cli estimatesmartfee nblocks=3
error: Error parsing JSON:nblocks=3
```

The second parameter has no named entry.

We are discussing this feature introduced in Version 0.14 because we thought they would press for all methods to have named parameters in Version 0.15.

The mempool

Our node or miner receives transactions from other nodes or miners. Our node then confirms every received transaction using a complex set of rules. Once a transaction is confirmed it is placed in an area of memory called the mempool. When a block gets confirmed, all the transactions present in it are removed from the mempool.

From Bitcoin Core version 0.14 onwards, each time bitcoin quits, it writes the mempool to disk in a file called mempool.dat. On startup, this mempool.dat file is read and the mempool area is populated. Earlier the mempool was never saved and hence between restarts, mempool transactions not placed in a block were lost for good.

Let's write code that reads the file, mempool.dat.

```
ch3606.py
from cfuncs import *
f = open("mempool.dat")
version = f.read(8)
print version.encode('hex')
number = r8bytes(f)
print "Number of Transactions %d:%x" % (number , number)
for tno in range ( 0, number):
    print "====Transaction Number %d:%d====" % (tno , f.tell())
    tver = rint(f)
    print "Transaction version %d" % tver
    sizeinputs = rvarint(f)
    print "Number of Inputs are %d" % sizeinputs
    witness = 0
    if sizeinputs == 0:
        witness = ord(f.read(1))
    print "Witness byte must be 1 %s" % (witness == 1)
    sizeinputs = rvarint(f)
```

```
print "The new number of inputs are %d" % sizeinputs
for i in range(0 , sizeinputs):
    outhash = rhash(f)
    print "Hash of transatcion bringing in outputs %s" % outhash[::-1].encode('hex')
    outindex = rint(f)
    print "Index of Output %d" % outindex
    lensig = rvarint(f)
    print "Length of signature %d" % lensig
    signature = rbytes(f , lensig)
    print "Signature %s" % signature.encode('hex')
    seqno = rint(f)
    print "Sequence Number %d" % seqno
    nooutputs = rvarint(f)
    print "Number of Outputs %d" % nooutputs
    for i in range(0 , nooutputs):
        bitcoinvalue = r8bytes(f)
        print "Output Number #%d" % i
        print "\tBitcoin Value %04.8f" % (bitcoinvalue * 1.0 / 100000000)
        lenscript = rvarint(f)
        print "\tLength of Script following %d" % lenscript
        script = rbytes(f , lenscript)
        print "\tScript %s" % script.encode('hex')

    if witness == 1:
        for inputs in range (0 , sizeinputs):
            print "Witness Number #%d" % inputs
            nowitness = rvarint(f)
            print "\tNumber of witnesses %d" % nowitness
            for wit in range(0,nowitness):
                bytestoread = rvarint(f)
                print "\tBytes to read %d:%x" % ( bytestoread , bytestoread)
                if bytestoread == 0:
                    continue
                data = rbytes(f , bytestoread)
                print "\t%s" % data.encode('hex')

            nLockTime = rint(f)
            print "nLockTime %d" % ( nLockTime )
            print "——where %d" % f.tell()
            time1 = r8bytes(f)
            print "Time %x" % time1
            nFeeDelta = r8bytes(f)
            print "nFeeDelta %d" % nFeeDelta
print f.tell()
```

Output

0100000000000000

Number of Transactions 11487:2cdf

The first 8 bytes of the file are for the mempool version number. Yes, it takes 8 bytes. When you are storing on disk, space savings do not matter. The version number in source code is defined in the following way.

```
./src/validation.cpp:4271:static const uint64_t MEMPOOL_DUMP_VERSION = 1;
```

The next 8 bytes are reserved for the number of transactions, we get a whooping 11487. Once again, a simple 4 bytes would suffice but they are thinking too much into the future. Then the for statement loops over every transaction.

There is no formal documentation of the file, mempool.dat. In fact, the documentation is given in form of code in the function LoadMempool within the validation.cpp file. Writing this program was like taking candy from a kid. Our trusted printf's displayed what is read from the file and the position of the file pointer.

For every transaction in the mempool, an entire transaction object gets written to disk. There is the transaction version number, followed by all the inputs. Then comes our witness friends. As a revision, because of past compatibility, the number of inputs will be 0 when this transaction carries witness data. That's why we check for the number of inputs being 0 and set the witness variable to 1. Then, the next byte read must be 1.

The count or the number of inputs is read once again using the same standard read functions from the file cfuns.py. All inputs from disk are read using a for loop. Next is the 32-byte hash bringing in bitcoins and then the 4-byte output index. Then the length of the signatures followed by the actual signature. Standard stuff. We end with the rarely used sequence number.

After the inputs is count of outputs, starting with the 8-byte Bitcoin value, then the length of the script and finally the script itself.

If there is a witness hanging around, the witness data is read. The count or number of witness records is not stored separately but it is equal to the number of inputs. The number decides the first for statement loop.

The number of witness records is stored in the variable nowitness. The first byte is the number of bytes following. If this byte has a value of 0, we do nothing and simply loop back. Otherwise, the witness data is read.

Finally, the last field of a transaction that everyone calls is nLockTime.

This ends one transaction's data. Then there are 8 bytes for the time and then something called nFeeDelta in the source code, which is always 0. Finally, there is a two-dimensional array called mapDeltas, which has a size of 0. This explains why the last byte of the file is 0. We could have understood what the last two fields do, but father time beckons.

We have seen the transaction data multiple times so it's not shown here. Just to confirm that we are on the right track, we display the last position in the file mempool.dat which must match the size of the file.

The added option in version 0.15 is called persistmempool. In the file init.cpp, we see these two lines.

```
if (fDumpMempoolLater && gArgs.GetArg("-persistmempool", DEFAULT_PERSIST_MEMPOOL)) {
    DumpMempool();
}
```

The function DumpMempool is called if the argument is true, which it is by default. The command line argument with bitcoind can be used to set it to false. Then the mempool will not be dumped to disk.

The function LoadMempool is called only once and from the file init.cpp. This is how it looks.

```
if (gArgs.GetArg("-persistmempool", DEFAULT_PERSIST_MEMPOOL)) {
    LoadMempool();
}
```

The same rules of the option persistmempool are followed. We wonder why these two if statements were not added in version 0.14.

Caching for Performance

We end our explanations on the new features of version 0.15 by talking about caching.

Let's start at the very beginning, So, who comes first, a transaction or a block? The answer is obvious, a peer first receives a transaction and then a block that may contain the transaction. When a peer receives a transaction, it conducts a series of checks to prove that the transaction is valid. The most time consuming of these checks is validating the signature. This check is slow only because Elliptic Curve Cryptography is very slow. The next slowest is the checking the script itself as hashing is also a slowpoke.

It's still not over after a transaction is checked once. This transaction at some point in time will be a part of a block. Each transaction in the block is also verified once again. This is a sheer waste of time, processing the same thing twice over.

So, in early days of Bitcoin, the signature validation was done only once and then a hash was created and cached. The next time around, the cache is checked first for the signature hash of the transaction. If present, then the signature is not validated again.

Henceforth, you will not see an output like ours unless you tinker with the source code. The idea is to get a feel for what caching is all about.

The new version, 0.15 added script validation to another cache. This eliminates the chance of doing the same CPU computations twice. The `printfs` in the `validation.cpp` file shows the following:

```
validation.cpp CheckInputs Start hash
83b32acf69fa8e48348309c5d9a28d378b818b0f6fb18735e8b31623c1e609ad
validation.cpp Adding to cache
badc10a096e4050e5b1cc65c765c53260b2d09e6373b3c2ccadbe247e552bd00
83b32acf69fa8e48348309c5d9a28d378b818b0f6fb18735e8b31623c1e609ad
```

When a transaction is received with a hash starting with 83b, a SHA-256 hash is calculated using a nonce or random number, the witness hash and flags. The hash value now begins with badc. Much later when we find this transaction in a block, it shows the following.

```
validation.cpp CheckInputs Start hash
83b32acf69fa8e48348309c5d9a28d378b818b0f6fb18735e8b31623c1e609ad
validation.cpp Cache Found
badc10a096e4050e5b1cc65c765c53260b2d09e6373b3c2ccadbe247e552bd00
83b32acf69fa8e48348309c5d9a28d378b818b0f6fb18735e8b31623c1e609ad
```

Since the hash starting with badc is already in the cache or buffer, it will get validated again, but the most expensive checks are not computed, the second time around.

```
Using 16 MiB out of 32/2 requested for signature cache, able to store 524288
elements
Using 16 MiB out of 32/2 requested for script execution cache, able to store 524288
elements
```

Here, the output of `bitcoind` educates us about an older signature cache and a newer script execution cache. The idea is to perform the expensive CPU computations just once and then cache this fact by storing a different hash each time in the cache. Before checking for any transaction, please check if the transaction is present in the cache. We avoid getting into the gory details on why this cache is called a cuckoo cache as it involves explaining lots of C++ code. But we will calculate the cached hash.

One major use of such caches is to prevent a Denial of Service attack. When a transaction is sent to a miner with a 1001 inputs, the signature validity will be checked a 1000 times. Then if the last input is invalid, the entire transaction gets dropped. A short while later, the same transaction comes in with 1002 inputs where the first 1000 inputs are the same. So, we run the same expensive checks again. But if the signature validation had been cached, these expensive checks can be avoided.

You may ask: Why not store all the 20 plus checks carried out on transactions? The answer my friend is that some validations depend on the block these transactions fall under. For example, the locktime field gives information on the block height a transaction will be valid from. The basic idea is that more the checks carried out once, the better the performance.

To square the circle, let's compute the hash that gets cached, which ultimately represents the script validated transaction. You will never see such a bitcoind output.

```
CheckInputs
CTransaction(hash=606c0949ffdf38f59288895504d993bb9ca57fd9184f9302caade59a
9cae9626, ver=1, vin.size=1, vout.size=2, nLockTime=0)
  CTxIn(COutPoint(1ee376306a, 0), scriptSig=00483045022100e72b59aca6)
  CScriptWitness()
  CTxOut(nValue=0.28777548, scriptPubKey=a914b9bfacbf1f4b7c5642a6e3cfe6a)
  CTxOut(nValue=0.00200000, scriptPubKey=a91498120c467cd9ff50c12a03410f)
Nonce
b03e58d995b485213e1a7734172cfe90e5bbdd7d79c41be4167af99e2d85438b:19
Witness Hash
606c0949ffdf38f59288895504d993bb9ca57fd9184f9302caade59a9cae9626
flags e15:3605:4
Write 8b43852d9ef97a16e41bc4797dddbbe590fe2c
Write 2696ae9c9ae5adca02934f18d97fa59cbb93d90455898892f538dfff49096c60
Write 150e0000
Write 80
Write 0000000000000001b8
Cache Value is 550be0536bb096e8a3e571b7dca8ea6850a38a266460a0e9747ec08384d4d191
```

```
ch3607.py
from cfuncs import *
nonce = "8b43852d9ef97a16e41bc4797dddbbe590fe2c"
witnesshash
"2696ae9c9ae5adca02934f18d97fa59cbb93d90455898892f538dfff49096c60"
flags = "150e0000"
s = nonce + witnesshash + flags
s1 = s.decode('hex')
temp = hashlib.sha256(s1).digest()
print temp[::-1].encode('hex')
550be0536bb096e8a3e571b7dca8ea6850a38a266460a0e9747ec08384d4d191
```

The CheckInputs function in the validation.cpp file calculates a hash cache for a transaction whose hash value starts with 606c.

The code then calculates a random number called nonce for which only 19 bytes of this value are used. As per the comments, it is because the SHA hash is calculated only once. The next value that gets added to the nonce is the

witness or transaction hash starting with 606c. So, the size of data to be hashed is $32 + 19 = 51$. Then, there is a 4-byte flag, thus totaling it to 55 bytes. There is a write, 0x80 holding a single byte and the length of 8 bytes, thus a total of 64 bytes. The last two writes are part of the hash's internal calculations.

We are not sure if the data bytes are in the reverse form, so we place a couple of printf's in the Write function of the SHA256 hash code. It's always better to see what bytes are used to compute this hash then speculate about it. The reverse bytes of the hash are displayed.

The nonce changes with every run of bitcoind and therefore the above program cannot be made generic. Plus, we have no access to the nonce. We could though write our own function that returns this nonce but its asking for too much at this stage.

Let's now turn our attention to the other cache, the signature cache, which is a little more complicated. The cache details are found in the file sigcache.cpp.

We are working with a transaction hash:

```
b844ef1d72d63b2f38c5e564f94b82c86e345e38a0dc086c11dfe16c795b088f.
```

The script sig of this hash is broken up into

Signature

```
47304402207b7b75886365ddd9618f12e4e41a1d58e66e9820c16f72b79ff023d976b097  
aa022076c831001d53c311a9f2a0b519efe0d66c38cc29395c8ee7977cf90f6a6718d701
```

Public key

```
21038e9942451fae2389d0e9ca420ed0b1b40d5ef8cf36fbbd014787787634cc18ac
```

ch3608.py

```
from cfuncs import *  
def reversehash(hash):  
    return "".join(reversed([hash[i:i+2] for i in range(0, len(hash), 2)]))  
  
nonce =  
reversehash("7ae724b1c71712fcfe33fd0e3a19b7a20d6e935e5f7d1f025162d61311f31387")  
hash =  
reversehash("962458a42a4614c1eadffc39c443b523d909cafd7052d4e3d31eed62c86f0097")  
pubkey =  
"038e9942451fae2389d0e9ca420ed0b1b40d5ef8cf36fbbd014787787634cc18ac"  
sig =  
"304402207b7b75886365ddd9618f12e4e41a1d58e66e9820c16f72b79ff023d976b097a  
a022076c831001d53c311a9f2a0b519efe0d66c38cc29395c8ee7977cf90f6a6718d7"  
s = nonce + hash + pubkey + sig  
s1 = s.decode('hex')  
temp = hashlib.sha256(s1).digest()  
print temp[::-1].encode('hex')
```

Output

```
3ff686c2efbc80aa9e11ad865d1c3c60581610167d73ebd30030c2857d111130
```

intepreter.cpp ChekcSig signash

```
962458a42a4614c1eadffc39c443b523d909cafd7052d4e3d31eed62c86f0097
```

Sigcache.cpp Nonce

```
7ae724b1c71712fcfe33fd0e3a19b7a20d6e935e5f7d1f025162d61311f31387
```



```

Sigcache.cpp hash
962458a42a4614c1eadffc39c443b523d909cafd7052d4e3d31eed62c86f0097
Sigcache.cpp Pubkey
038e9942451fae2389d0e9ca420ed0b1b40d5ef8cf36fbdd014787787634cc18ac
Sigcache.cpp vchSig
304402207b7b75886365ddd9618f12e4e41a1d58e66e9820c16f72b79ff023d976b097aa
022076c831001d53c311a9f2a0b519efe0d66c38cc29395c8ee7977cf90f6a6718d7
Sigcache.cpp Entry
3ff686c2efbc80aa9e11ad865d1c3c60581610167d73ebd30030c2857d111130

```

The hash to be added to the signature validation cache starts with 3ff686. This variable that stores the cache hash is called entry in the original source code. It is made up of 4 different components.

The first is called the nonce which is random and reversed and changes each time. You cannot run this program because of the nonce changing. The last two fields are obtained from the input scriptSig, the value is broken up into an actual signature and a public key. The starting byte 0x47 and the last byte 01 of the signature are not written. For the public key, the first byte 0x21 is not written.

The hash once again is not a normal transaction hash, it is very different. The obvious thing to do is simply use the transaction hash, but a special class called CTransactionSignatureSerializer was created with some changes. This had to be done some day as this signature hash was to be done in-place, straight from the comments in the code. This new hashing code is in the interpreter.cpp file.

This signature validation cache hash is old, the newer cache hash looks cleaner. It will be difficult to understand the above. The excitement comes in when you calculate the hashes that go into these two caches.

Pruning a Blockchain

This section has nothing to do with changes in Bitcoin version 0.14 or 0.15. We never pruned a blockchain before, so we thought we could share our experience. We first deleted or renamed the Bitcoin folder to start from scratch.

The bitcoind command changes as shown below.

```
bitcoind -printtoconsole -prune=1
```

The command line option prune with a value of 1 does effectively nothing. It does not delete any blocks, it simply allows us to prune our blockchain later through the bitcoin client.

```

2017-09-26 13:56:34 UpdateTip: new
best=000000000000000000000000401f42637257e6514c4ef4821cf3c494f252bcc9509d1b
height=397390

```

When we run the client, bitcoin-cli, bitcoind responds to this connection with the height of our last block. The last block was at height 397390. This value has no significance at all.

Now the next command is to prune the blockchain.

```

bitcoin-cli pruneblockchain 397388
397102

```

The pruneblockchain method needs a block height. Ours is close to the actual block height. The height value returned is 397102 which is very close to our height value.

The output we see in bitcoind is shown below.

```
2017-09-26 13:56:44 Prune: UnlinkPrunedFiles deleted blk/rev (00434)
2017-09-26 13:56:44 Prune: UnlinkPrunedFiles deleted blk/rev (00435)
```

The method `pruneblockchain` deletes blk and rev files. This is what the blocks folder looks like

```
-rw----- 1 vijaymukhi staff 133513678 Sep 26 18:58 blk00436.dat
-rw----- 1 vijaymukhi staff 133713090 Sep 26 19:13 blk00437.dat
-rw----- 1 vijaymukhi staff 18874368 Sep 26 19:13 rev00436.dat
-rw----- 1 vijaymukhi staff 17825792 Sep 26 19:23 rev00437.dat
```

The first file is not `blk00001.dat` but `blk00436.dat`. Ditto for rev files. All our Bitcoin blk and rev files have been deleted in one go. Our Bitcoin folder is a measly 2.49 GB from the peaked 190 GB. Most of this space is occupied by the `chainstate` folder at 1.87 GB followed by the `blocks` folder at 612 MB. The index folder within blocks is now 76MB as the transaction indexing is off. Something to remember, you cannot prune when the transaction indexing is on. It should come as no surprise now that this book took so long to write.

The question is why did the pruning stop before the specified block number.

```
2017-09-26 13:43:36 Pre-allocating up to position 0x3000000 in blk00438.dat
2017-09-26 13:43:37 UpdateTip: new
best=000000000000000001e7737a23153d52052550a29abd5c6fc54de9e7b7c04ed3
height=397102
```

The above bitcoind output shows us that block number 397102 is the first block in the file `blk00438`. The file numbers `blk00436` and `blk00437` are retained since they are too close to the tip. All blocks are not deleted, some blocks before the specified height are retained, just in case we may want to read some previous block data.

There is a variable called `MIN_BLOCKS_TO_KEEP` with a value of 288. The height returned by bitcoind is 397390. So, when we subtract 288 from this height, it gives us a height of 397102. Once you prune, you cannot go back. So, it means starting afresh and re-downloading the block chain. Some things you cannot undo.

Some errors that we encountered

Cannot prune blocks because node is not in prune mode

We must start bitcoind with the prune option set to 1.

Blockchain is shorter than the attempted prune height.

This error comes about when the prune height specified is larger than the last block height in our blockchain.

The minimum height of the blockchain must be larger than 100000 blocks on mainnet and 1000 blocks on testnet and regtest. The files are deleted in lock-step, first the blk file and then the rev file of the same number.

```
bitcoin-cli getblockhash 1
00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048
```

All commands run as normal. Here we are retrieving the hash value of block 1. We get no errors, But.

```
bitcoin-cli getblock 00000000839a8e6886ab5951d76f411475428afc90947ee3201
error code: -1
error message:
Block not available (pruned data)
```

The block data is not available because the `getblock` method reads the `blk00000.dat` file to get this data. On the other hand, the `getblockhash` method works because it gets the block hash from the index folder. It does not read the dat files stored on disk.

We do not prune even though only a few methods do not work. We also cannot use methods like `gettxout` because the transaction indexing must be disabled before pruning.

Now we did something tricky. We deleted the `blk00436.dat` file from the blocks folder. Immediately, we get an error stating all blocks are not present on disk. So, we wondered, how does bitcoind know about the files we pruned. On our disk, we have only the chainstate and the index folder. And we checked, no new key in the chainstate database. The only culprit is the index folder. We ran the same code that displays a key called `f`. And we get the following:

```
66b4010000
Block Number 436
804abed4824e87d5db27979a60979d3384b4cfbf3584b4d8de69
Number of blocks in the file is 202
Block File Size is 133513670
Rev Block Size is 18198055
Lowest Block Number in file 396768
Largest Block number in file is 397107
Smallest Time in Block is Fri Feb  5 09:38:21 2016
Highest Time in Block is Sun Feb  7 03:42:57 2016

66b3010000
Block Number 435
0000000000000000
Number of blocks in the file is 0
Block File Size is -8
Rev Block Size is 0
Lowest Block Number in file 0
Largest Block number in file is 0
Smallest Time in Block is Thu Jan  1 05:30:00 1970
Highest Time in Block is Thu Jan  1 05:30:00 1970
```

You can refresh your memory by re-reading the chapter on the index database. The file with block number 435 has all 0's as the value for all its fields. Even the date is zeroed out. It indicates that a block previously present at this location is now pruned. Similarly, block 436 has all its fields filled up with good values.

The `l` key returns the last file number in our blockchain.

```
6c
b7010000
439
```

So, block 439 is the largest block, and going backwards we can figure out with the `f` key the number of files that make up the blockchain.

We are good at giving unsolicited advice but no one takes us seriously J

CHAPTER 37

Transactions and Blocks - Error Checks

This is the last chapter in the book having python code. We have come a full circle. We plan to revisit the basics of the Bitcoin blockchain in this chapter. The modus operandi here is to read the Bitcoin source code written in C++ and convert it into python code. The code segment chosen is the one that checks for errors in a transaction and a block. It will be one more way to observe how the transactions and blocks are structured in a blockchain. This transaction validating code is spread over many files. The variables are given the same names in the program as in the original source.

The file `net_processing.cpp` receives a transaction from the Bitcoin network. Then this newly minted transaction is exiled to the mempool. But prior to that, a battery of error checks take place. Once the transaction passes these error checks, it is added the mempool. The transactions in mempool are valid but no miner has the guts to place them in a block.

After a wait of ten minutes or longer, some miner gets lucky and successfully creates a block. This block is then sent to anyone who is willing to accept it. The transactions added to the block are removed from the mempool. Your mempool may not have all the transactions shifted to a block. The rule is : All error checks are performed on transactions before they enter the mempool.

We learnt a lot on different error checks performed on a transaction by reading the source code. It ultimately sinks in that a Coinbase transaction is never ever sent by a miner to other peers. A Coinbase transaction is only found in a block, it will never ever be found in a mempool. A Coinbase transaction is very different. It's obvious now after reading the comments, but not before. A handful of transaction error checks are covered as our self-imposed deadline was looming large.

```
ch3701.py
import subprocess
import json
def GetOp2(scriptSig):
    opcode = ord(scriptSig[0])
    nSize = 0
    cnt = 0
    error = 0
    if opcode <= OP_PUSHDATA4:
    if opcode < OP_PUSHDATA1:
        nSize = opcode
        cnt = 1
    elif opcode == OP_PUSHDATA1:
        nSize = ord(scriptSig[1])
        cnt = 2
```

```

if len(scriptSig) < 1:
    error = 1
elif opcode == OP_PUSHDAT2:
    nSize1 = ord(scriptSig[1])
    nSize2 = ord(scriptSig[2])
    nSize = 1 * nSize1 + 256 * nSize2
    cnt = 3
    if len(scriptSig) < 2:
        error = 1
    return ( nSize , scriptSig[nSize + cnt:] , error)
def IsPushOnly(scriptSig):
    scriptSig = scriptSig.decode('hex')
    if ord(scriptSig[0]) == 0:
        scriptSig = scriptSig[1:]
        while len(scriptSig) > 0:
            (len1 , scriptSig , error ) = GetOp2(scriptSig)
            if error == 1:
                print "This transaction is not pushonly:scriptsig-not-pushonly"
            if len(scriptSig) != 0:
                print "The length of the scriptSig must be 0"
def outputtype(scriptPubKey):
    if len(scriptPubKey) == 23 and ord(scriptPubKey[0]) == 0xa9 and
        ord(scriptPubKey[1]) == 0x14 and ord(scriptPubKey[22]) == 0x87:
        return 'scripthash'
    elif len(scriptPubKey) == 34 and ord(scriptPubKey[0]) == 0x00 and
        ord(scriptPubKey[1]) == 0x20:
        return "witness_v0_scripthash"
    elif len(scriptPubKey) == 22 and ord(scriptPubKey[0]) == 0x00
        and ord(scriptPubKey[1]) == 0x14:
        return "witness_v0_keyhash"
    elif len(scriptPubKey) >= 1 and ord(scriptPubKey[0]) == 0x6a :
        return "TX_NULL_DATA"
    elif len(scriptPubKey) == 25 and ord(scriptPubKey[0]) == 0x76 and
        ord(scriptPubKey[1]) == 0xa9 and ord(scriptPubKey[-2]) == 0x88 and
        ord(scriptPubKey[-1]) == 0xac:
        return "pubkeyhash"
    elif len(scriptPubKey) == 35 and ord(scriptPubKey[0]) == 0x21 and
        ord(scriptPubKey[-1]) == 0xac :
        return "pubkey"
    elif ord(scriptPubKey[-1]) == 0xae:
        m = ord(scriptPubKey[0]) - 80
        n = ord(scriptPubKey[-2]) - 80
        length = len(scriptPubKey) - 3
        pubkeylen = n * 33 + 3
        if m < 1 or n < 1 or m > n or pubkeylen != length:
            return "Non-Standard"

```

```
    else:
        return "multisig"
    else:
        return "Non-Standard"
sighash = [
    'fc90816cc18544e856e2550e321081ef2ce2a9ff3e41c636041bfbdad4c53943' ,
    #normal
    'a8d0c0184dde994a09ec054286f1ce581bebf46446a512166eae7628734ea0a5' ,
    #coinbase
    'e79f67a394acf3220f43025db58478dc19e1ef93043ec8d9a71747cfe4b4dee7' ,
    #witness
    '552026dade1c9385e4693a4e82f07080d8d1950fc822346f95a0dc1e0a833465' ,
    #multisig
    '2616c318f15ff1e696f71c253d0ccea615b4e0ebb4c9631ed5e4de155262c3db' ,
    #multig small
    '90154afd25089357f1cc7f9bd300781f903a1afec5a1e19ae189b902fca84b6b' ,
    #OP_RETURN
    'da738e29f64e90ae46dcc3e6b4154041d6324abbe7919e722d486a4a3148b7dc' ,
    #pushonly no opcode ae at end
    'c0334ba7f58ce1701718ce4f5ae5a51eff304de7e2771d9c6305f6eaca35ff12' ,
    #pushonly for PUSHDATA1
    '552026dade1c9385e4693a4e82f07080d8d1950fc822346f95a0dc1e0a833465' ,
    #pushonly for PUSHDATA2
    'aed8756b3d299d67e8d2af7d5008857d5fde5d0007a4535641f70977d29633cd' ,
    #TX_SCRIPTHASH
    '90154afd25089357f1cc7f9bd300781f903a1afec5a1e19ae189b902fca84b6b' ,
    #OP_RETURN + pubkeyhash
    '38400c638473f08fffd3f42442afa379405444ef729e74fd67b13f13f56717cd' ,
    # TX_PUBKEY + pubkeyhash
    '56214420a7c4dcc4832944298d169a75e93acf9721f00656b2ee0e4d194f9970' ,
    #multisig output
    'da738e29f64e90ae46dcc3e6b4154041d6324abbe7919e722d486a4a3148b7dc' ,
    #multisig with no ae
    'af32bb06f12f2ae5fdb7face7cd272be67c923e86b7a66a76ded02d954c2f94d' ,
    #non standard
    'c0b2cf75b47d1e7f48cdb4287109ff1dd5bcf146d5f77a9e8784c0c9c0ef02ad' ,
    #non standard
]
for total in range(0, len(sighash)):
    raw = subprocess.check_output(["bitcoin-cli" , "getrawtransaction" ,
    sighash[total] , "1" ] )
    print sighash[total]
    #print raw
    raw = json.loads(raw)
    coinbase = ''
    try :
        coinbase =raw['vin'][0]['coinbase']
```

```

except:
    pass
#CheckTransaction Function in tx_verify.cpp
if len(raw['vout']) == 0:
    print "This transaction comes with no outputs:bad-txns-vout-empty"
if len(raw['vin']) == 0:
    print "This transaction comes with no inputs:bad-txns-vin-empty"
WITNESS_SCALE_FACTOR = 4
MAX_BLOCK_WEIGHT = 4000000
if raw['vsize'] * WITNESS_SCALE_FACTOR > MAX_BLOCK_WEIGHT:
    print "The transaction is too large:bad-txns-oversize"
COIN = 100000000
MAX_MONEY = 21000000 * COIN
nValueOut = 0
for o in raw['vout']:
    if o['value'] < 0:
        print "The transaction Bitcoin value cannot be negative:bad-txns-vout-negative"
    if o['value'] > MAX_MONEY:
        print "The transaction Bitcoin amount cannot be greater than
MAX_MONEY:bad-txns-vout-toolarge"
    nValueOut = nValueOut + o['value']
if not (nValueOut >= 0 and nValueOut <= MAX_MONEY) :
    print "The total value of all the Bitcoins in the output also cannot exceed
MAX_MONEY:bad-txns-txouttotal-toolarge"
if len(coinbase) == 0:
    list1 = []
    for i in raw['vin']:
        both = i['txid'] + str(i['vout'])
        list1.append(both)
    list2 = set(list1)
    if len(list1) != len(list2):
        print "We have a duplicate output that we are refering to in the inputs:bad-txns-inputs-duplicate"
if len(coinbase) != 0:
    coinbase = coinbase.decode('hex')
    if len(coinbase) < 2 or len(coinbase) > 100:
        print "Coinbase is too small or too large:bad-cb-length"
    else:
        for i in raw['vin']:
            if i['txid'] == '0000000000000000000000000000000000000000000000000000000000000000'
            and i['vout'] == 4294967295:
                print "Cannot have a null transaction hash:bad-txns-prevout-null"
        witness = 0
        for i in raw['vin']:
            try:
                i['txinwitness']

```

```
witness = 1
except:
    pass
    #if witness == 1:
    # print "If Witness is found then check if witness was activated:no-witness-yet"
    #IsStandardTx policy.cpp
    MAX_STANDARD_VERSION=2
    if raw['version'] > MAX_STANDARD_VERSION or raw['version'] < 1:
    print "Wrong Transaction version: version"
    WITNESS_SCALE_FACTOR = 4
    MAX_STANDARD_TX_WEIGHT = 400000
    sz = 4 * (raw['vsize'] - 0.25 )
    if sz > MAX_STANDARD_TX_WEIGHT:
    print "Transaction size too large:tx-size"
    for i in raw['vin']:
    a = i.get('scriptsig', '')
    if len(a) >= 1:
    a = a['hex']
    if len(a)/2 > 1650:
    print "The script signature is too large: scriptsig-size"
    DUST_RELAY_TX_FEE = 3000
    for i in raw['vout']:
    nSize = (len(i['scriptPubKey']['hex']) + 16 + 2) / 2
    nSize = nSize + ( 32 + 4 + 1 + 107 + 4)
    dust = DUST_RELAY_TX_FEE * nSize / 1000
    if not i['value'] < dust:
    print "The dust is too large:dust"
    MAX_SCRIPT_SIZE = 10000
    for i in raw['vout']:
    opcode = ord(i['scriptPubKey']['hex'][0:2].decode('hex'))
    if opcode == 106 and len(i['scriptPubKey']['hex'])/2 < MAX_SCRIPT_SIZE:
    print "This is OP_RETURN unspendable transaction or it is too large:"
    OP_PUSHDAT1 = 0x4c
    OP_PUSHDAT2 = 0x4d
    OP_PUSHDAT4 = 0x4e
    for i in raw['vin']:
    a = i.get('scriptSig', '')
    if len(a) >= 1:
    a = a['hex']
    if len(a) >= 1:
    IsPushOnly(a)
    noopreturn = 0
    DEFAULT_PERMIT_BAREMULTISIG = True
    for i in raw['vout']:
    otype = outputtype(i['scriptPubKey']['hex'].decode('hex'))
    if otype == 'TX_NULL_DATA' :
```



```

noopreturn = noopreturn + 1
if otype == 'multisig' and not DEFAULT_PERMIT_BAREMULTISIG:
    print "multisig:bare-multisig"
if otype == 'Non-Standard':
    print "A non-standard transaction:publickey"
if noopreturn > 1:
    print "Only one OP_RETURN per transaction:multi-op-return"

```

Output

```
90154afd25089357f1cc7f9bd300781f903a1afec5a1e19ae189b902fca84b6b
```

This is an OP_RETURN unspendable transaction which is quite large:

A transaction object is made up of 4 different types of entities. There is a version number, then a series of inputs followed by a series of outputs and finally, a locktime. In C++ code, a CTransaction class represents a transaction. The inputs and outputs are vectors or lists as they can store multiple values. A reminder again that a vector in C++ starts with a count and then with the actual objects.

With segregated witnesses activated, the new serialization is version, then vinDummy or a 0 followed by a 1, then the lists of inputs and outputs followed by the witness data, if any and lastly the locktime.

The outputs are very simple. Simply, an 8-byte Bitcoin value and the scriptPubKey which starts with the length of key. The inputs are more complex, they start with a transaction hash and an output index as one object. There is a COutPoint in the C++ source followed by the signature+public key and the sequence number. For some reason, the witness data is part of the input's class but written after the outputs.

We could have used the getrawtransaction method with the transaction hash to get the raw bytes of the transaction and then written our own python code to create objects like CTransaction and more. But we prefer living in our comfort zone now. So, we use a json like formatted output to create the python objects, which normally are in sync with the internal C++ transaction objects.

The Transaction Error Checks

A list called sighash is created instead of individual variables to store multiple transactions. The first transaction in this list has many inputs and outputs. The second thash belongs to a Coinbase transaction. The third is a segregated witness transaction. We know this already because we tested our code many times to locate such transactions. Plus, these transactions must be vintage October 2017 also.

The variable coinbase stores the Coinbase transaction script, if any. If you have forgotten only the 0th input of the first transaction in a block is called a Coinbase transaction. The field is also named coinbase as seen in the output. This check is required when we are checking for a Coinbase transaction in a block.

The error checks are in the same order as in the C++ source code.

The first three checks do not depend upon context. This line is copied verbatim from the comments in the source code.

Check #1: No valid transaction can have 0 outputs. So, we first check if there is at least one Bitcoin output or address to whom we can transfer the bitcoins. While displaying, the first error message is our text and the message after the colon is taken from the source code. The list raw['vout'] must have a minimum length of 1.

Check #2: The output in some transaction brings in the money or bitcoins. So, the second check is for the transaction to have at least one input. The length of the inputs is 0 for a segregated witness transaction. Here we check if the size of the list raw['vin'] is 0.

Check #3: The third check is on the size of the transaction. In segregated witness, the size is replaced by a weight. The maximum block weight is 4MB or denoted by the macro/#define/variable `MAX_BLOCK_WEIGHT`. The size field is always larger than the vsize field as it represents the size taken by the entire transaction. The vsize field is the newer transaction size that ignores the witness data.

The transaction size is checked to be less than 4MB. This check is performed without considering the result of the witness data multiplied by the witness scale factor, 4. In other words, if there were no witnesses lurking around, the size of a single transaction would be 1 MB, like before. By having witnesses around, the data size in a block increases but the witness can still be ignored.

We understand the comment that says, 'that doesn't take witness into account', but what follows next is beyond us, 'as that hasn't been checked for malleability'. We assumed that after the onset of segregated witnesses, transaction malleability was a topic for the history books.

The next three checks are for the value of Bitcoins.

Check #4,5,6: The fourth check cannot get simpler, there cannot be a negative amount, 0 is allowed as we have seen before. The fifth check says that you cannot have an amount larger than a number with zeroes beyond a limit. In the sixth check, the running total, which is sum of the amounts in all the outputs must be positive and less than `MAX_MONEY`.

Check #7: Now for the seventh check. We learnt a long time back that a Bitcoin output is atomic. So, either you spend all the bitcoins in an output or none. The lifecycle of an output starts when someone sends your Bitcoin address some money, this record is part of the UTXO set. In future, some input in a transaction references this specific output, which is then removed from the UTXO set. There cannot be any two similar inputs in the same Bitcoin transaction or any other bitcoin transaction having the same combination of transaction hash and output index.

The seventh check confirms that the transaction hash and output index are not the same across all inputs in a transaction. The Coinbase transaction is exempted as it has only one input which can be anything.

The constant 32-byte transaction hash and the output index are concatenated and then added to a list called list1. A set object is created to remove the duplicates. If the length of the two lists are different, then there are two or more inputs that point to the same output in a transaction hash. We can reference two different output indexes in the same transaction.

The C++ code is much cleaner in this situation as we cannot even call our code a kludge. According to the comments, this is a very slow check and hence it is carried out only when a variable called `fCheckDuplicateInputs` is used.

Check #8: The length of the coinbase field must be larger than 2 bytes and less than 100 bytes. No point asking why 100, as the Coinbase transaction input brings nothing to the table of reason.

Check #9: The transaction hash in the input (if not a Coinbase transaction) must point to a valid transaction carrying outputs. A hash with all 0's is a no-no. This hash, also called a null hash in the class `uint256`, is only valid in a Coinbase input. The next check is to make sure that the output index value is not equal to 4294967295, which is the largest value a 32-bit unsigned integer can hold minus 1.

Check #10: The file called `versionbits.cpp` makes known whether segregated witnesses are activated or not. This value is hard coded in the source code. The activation does not rely on our active blockchain tip being larger than a certain predefined block height. There is a field in the inputs called `txinwitness` to check for a segregated witness. In the C++ source code, things are done differently.

Check #11: The transaction version is checked to have one of the two values valid today, 1 or 2, the value of the macro `MAX_STANDARD_VERSION`. This macro is meant for the transaction version not the block version.

Check #12: The Transaction Weight cannot be larger than 4 MB. We first compute the transaction weight using the formula given below.

```
#./src/core_write.cpp:163: entry.pushKV("vsize", (GetTransactionWeight(tx) +
WITNESS_SCALE_FACTOR - 1) / WITNESS_SCALE_FACTOR);
```

The vsize member in the json transaction output gives the Transaction Weight, it is assigned to the sz variable. The difference between size and vsize has been explained earlier. The condition check here is that our newly minted variable sz cannot be larger than 4 MB.

Check #13: This one is a masterpiece. The length of the scriptSig or the signature+publickey bytes cannot be larger than 1650.

Let's find out where this magic number 1650 comes from.

From Bitcoin Core Version 0.10, the transaction input can hold a maximum of 15 of 15 P2SH multisig transaction type. This type is written as m of n. This validity rule applies for compressed keys and 7-of-7 for uncompressed keys, which are larger in size.

The validity rules kick in again, the redeemScript that we calculated must have a maximum size of 520 bytes. A redeemScript looks like: m pub1 pub2 ... n OP_CHECKMULTISIG. The m and n and the last opcode take up 3 bytes, so only 517 bytes are left for the public key. A ECDSA public key compressed is 33 bytes large which multiplied by 15 resulting in 495, but when multiplied by 16 the answer is 528 bytes. This explains the 15 by 15. An uncompressed public key is 66 bytes large and when multiplied by 7, the value is 462, well within our limit of 517. But, 66 multiplied by 8 gives a size of 528, too large. This explains the 15 and 7.

The code shown in comments does not have the luxury of being wrong. It includes the size or length byte also. So, it is $(33 + 1)$ or 34 multiplied with 15 and then 3 bytes are added $(34 * 15) + 3$. The final length is 513 bytes. This explains the 513 bytes in the comments present in the file policy.cpp in the policy folder.

Another revision:

When you look at the bytes using the last thash value, you will notice that the script signature starts with a 00 byte. Now comes the size of the signature, which in Elliptic Curves can be 71 or 72 or 73 bytes. The probability of each size is 25%, 50 % and 25 % respectively. Taking a maximum size of 73 bytes and having 15 signatures as the maximum, the signatures takes up $73 * 15 = 1095$ bytes + 513 = 1608 bytes. An extra byte is added for the PUSHDATA opcode. This results in a size of $1608 + 15 + 1 = 1624$.

After the signatures, there are three bytes to account for. The opcode 4d stands for a PUSHDATA2 where we read the next two bytes in our case 01 02, to give us the size of the public key data following. One more reason for the maximum length of the script signature to be 1627, rounded off to 1650 to future proof the error check.

All the calculations are from the comments, not our thinking. Our book has transpired thanks to the comments in the source code. Here, they give information of a non-standard 20-of-20 CHECKMULTISIG scriptPubKey.

Revisiting the source code clarified this specific point that a transaction input cannot be 1627 bytes large. Basically, putting everything together.

Check#14. This check is a dusty one and performed on every output. First, a Dust Threshold is calculated. So, we first calculate the size of an output and store it in a variable, nSize. The Bitcoin value will always take 8 bytes or 16 digits but the actual Script is variable nSize. It's normally 25 bytes long plus one byte for the length. This variable nSize, the size of the output will normally be $(26+8)$ 34 bytes large.

Now we do something different. We take the theoretical size of an input. A sum of the transaction hash 32 bytes, 4 bytes for the output index, 1 byte for the signature length, 107 bytes for the signature and finally 4 bytes for the sequence number. This size is 148 bytes. Signatures can be 1627 bytes large also. The total size is $148 + 34 = 182$ bytes.

Now we calculate the dust threshold which is the size of the output and input multiplied by a constant `DUST_RELAY_TX_FEE` or 3000 and then divided by 1000. Our dust threshold is a measly 546. The condition is that the value of the Bitcoins in the output must be greater than dust.

We would like to mention that if we are on a transaction which has a segregated witness, we divide the above 107 by a value `WITNESS_SCALE_FACTOR`, which we mentioned earlier has a value of 4. This could only mean that we are only using 25% of the signature length and not all of it. A good saving if the size is at a maximum of 1627 bytes. The dust is measured as Satoshi per KB.

Now to understand the all the fuss about defining dust. Dust happens when you pay more fees than the Bitcoin value in the output.

Let's assume that it costs us 3000 Satoshi per kilobyte to relay bitcoins. We just computed the size of the simplest transaction at 182 bytes, so the fee in Satoshi is $182 * 3000 / 1000$ or 546 Satoshi. The 100 is to convert KB into bytes. If the amount is less than this number, it means that we are spending more in fees than the actual Bitcoin value of a specific transaction output. This also means that the minimum Bitcoin output value must be 546 Satoshi.

On the other hand, the size of a single transaction output drops to 31 bytes for a segwit transaction. The segwit output starts with a `OP_HASH160` opcode and ends with a `OP_EQUAL`. The actual output size should be 32 bytes as we have 2 opcodes less and not 31 bytes. Please check the output yourself, the length byte which was 0x19 now drops to 0x17. So, the value should be 68 and not 67, but no one dare argue with the Bitcoin Gods. We are not quibbling over 1 assumed byte less. The assumed input size is now 67 bytes as 107 divided by 4 is 26. The transaction size is now 98 bytes. $98 * 3000 / 1000 = 294$ Satoshi. This is our calculation, the comments only calculated the value of 546.

Check #15: Here we check if our transaction output is spendable or not. We learnt earlier that a transaction output can start with the opcode `OP_RETURN` or opcode 106 and thereafter any junk or data is given. This opcode opens the option of filling the blockchain with whatever junk we want. This check is on such transactions where the output starts with a `OP_RETURN`. The outputs placed after this code may have a Bitcoin value but these are un-spendable outputs and have no place in a UTXO set. These are not invalid outputs so there are really no error messages from the Bitcoin Core. An additional check is performed on the maximum script size (10,000) bytes, too much for our liking but who listens. An `OP_RETURN` opcode does not make a transaction bad so it is not rejected and hence the check.

Check #16. This one check gave us sleepless nights as even Google has not heard of a Push only transaction. Let's look at the bigger picture of scripts. Bitcoin scripts are made up of simple bytes from 0 to 255. So, a number 21 in a script could either mean a byte in the public key or it could mean push 21 bytes on the stack. The meaning of a script byte depends upon context. Numbers up to 0x4b refer to the number of bytes to be pushed on the stack. The opcode 0x4c or `PUSHDATA1` indicates that the next byte will tell how many bytes are to be pushed on the stack. We can push 255 bytes on the stack with this opcode.

The opcode `OP_PUSHDATA1` uses the next byte to specify how many bytes should be pushed on the stack, it behaves like the single length byte. If there are more than 256 bytes to be pushed, the opcode `PUSHDATA2` is used where the next two bytes decide how many bytes are needed. This is more than what a valid script can handle. Hence the introduction of opcode `PUSHDATA4`, which is never used as 4 bytes are needed for the length of data to be pushed. This breaks all the script constraints.

We check if there is say an opcode `PUSHDATA2`. If yes, then the next two bytes will give the length of the following data. If not, then the script ends with an opcode `PUSHDATA2`. There is no space even for the length bytes. This script is invalid. Ditto for an opcode `PUSHDATA1` where 1 byte is required after the opcode.

Coming to the code, every input is scanned and given to the `IsPushOnly` function. The script bytes are stored in the hex field of the inputs. The function and function names are taken from the source code. The first check is for a 00 byte, which must be skipped. As before, the read byte is checked to be a byte that pushes data, if yes than that many bytes are skipped. The function `GetOp2` performs this task.

The function `IsPushOnly` scans the entire script. The error variable is given a value of 1 or an error is flagged when an opcode of 4c or 4d has no bytes, 1 or 2, left in the script. An opcode less than 4c is not considered. This condition is not tested as we have still to come across a transaction that is push only. But we tested our code on three different transaction hashes. Someone should document the error checks at the bare minimum. We need examples where transactions inputs are push only to cross check our code.

Check #17,18,19: This check is for outputs only. There are 8 different types of output. We have a transaction for each output type. Before a transaction output type is declared as non-standard, we check on all the types and perform these error checks. Check the newer UTXO set for another explanation.

The first output type is called `TX_SCRIPTHASH` and it is denoted by a number, 3 in the source code. The source code uses a function called `Solver` which is in a file `standard.cpp`. This function returns the output type and verifies if it is a standard transaction or not. A function called `IsPayToScriptHash` in the source code checks for the output size to be exactly 23 bytes long. Here the first byte is the opcode `OP_HASH160`, followed by the size of the hash 20 bytes and finally the last byte being `OP_EQUAL`.

The function `outputtype` performs the same task. The comments suggest that this output type is bounded by maximum constraints. We will always end with an example so make sure that you run the code with the right hash. We will give you a transaction hash and a type to double check.

```
a9141c1788a9b110587dff5f3e2eb03770ee3c10c7f787 scripthash
```

The second and third output types are part of the witness relocation program (one more joke) . The second one is called `witness_v0_scripthash`, here the size is 34 bytes. The first byte is 0 and it is called `witnessversion`. As an added check, the second length byte must be 32 bytes large.

```
0020701a8d401c84fb13e6baf169d59684e17abd9fa216c8cc5b9fc63d622ff8c58d
witness_v0_scripthash
```

The third output type is called `witness_v0_keyhash`. The length is smaller at 22 bytes and the first byte again is 0, the same `witnessversion`. The length or the witness program size must come in at a smaller 20 bytes.

```
0014e200dde45eb0529aeb8e16060fb9b109008b56
```

We realized that this transaction was later not present anywhere in the mempool or blockchain. So, we leave it here.

The fourth output type is called `TX_NULL_DATA` by the source code and we call it a `OP_RETURN`. This check is very simple, if the first byte is `OP_RETURN`, the length must be larger than 1. This rest of output must not be a Push Only. Please note that the earlier Push Only check was for inputs and this check is for outputs. We ignore the Push Only check.

```
6a0a56696a61794d756b6869 TX_NULL_DATA
```

The fifth output type is the most common of all, the `pubkeyhash`. The length is 25 bytes, the first byte is the opcode `OP_DUP` followed by the opcode `OP_HASH16`, then the length byte of the hash, 20 bytes. The second last byte is the opcode `OP_EQUALVERIFY` and the last opcode and byte is `OP_CHECKSIG`. We have a 20-byte hash, one length and 4 opcodes.

```
76a914f0ead6281c5a2fb5cb2dfbc57e8d43b134d6090188ac pubkeyhash
```

The sixth opcode type is `pubkey`. Only three checks, the length must be 35 bytes only, now the length is the first byte at 33 bytes, and the last byte is the single opcode, `OP_CHECKSIG`.

```
21029acf1dcd9f5ff9c455f8bb717d4ae0c703e089d16cf8424619c491dff5994c90ac
pubkey
```

Let's look at the most difficult one of all the output types, multisig. It was complicated when we looked at in the inputs, and now it will be a problem in the outputs also. Yes, the same multisig or opcode ae exists in the outputs also. The error checks are different. We check if the last byte is a OP_CHECKMULTISIG. The second last byte is a number that gives us n, the total number of keys. The first byte is m, the number of keys to sign. These numbers are subtracted from 0x50 to arrive at the final answer. There is one more check, which makes sure that there are n keys in the output. The length variable accounts for the size occupied by the public keys only, thus ignoring the first and last two bytes. The variable pubkeylen is also the size of the public keys, assuming there is a fixed size of 33 bytes plus the three length bytes, 0x21.

A transaction can be valid with multiple keys without an opcode ae sitting at the end. We need a separate book to write on the differences between a m of n multisig, multisig output versus a P2SH multisig. P2SH multisig represents a multisig script inside a Pay to Script Hash (P2SH). It took a lot of Googling to find an output of this kind.

```
522102c08786d63f78bd0a6777ffe9c978cf5899756cfc32bfad09a89e211aeb926242210
33e81519ecf373ea3a5c7e1c051b71a898fb3438c9550e274d980f147eb4d069d21036d5
68125a969dc78b963b494fa7ed5f20ee9c2f2fc2c57f86c5df63089f2ed3a53ae multisig
```

The eighth or the last error check is the catch all. If the transaction does not meet any of the above seven criterions, it is termed as a non-standard one. There are too many of them to show you, hence we restrict ourselves to two non-standard transactions only. Please do not try to decipher them.

```
aa20000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f87
NonStandard

0e54686543616b654973414c69650a7576a914d92d8d187f3bf869c1695bba74a9c57a4a84f0b188ac
NonStandard
```

The million-dollar question, what is all this leading to?

The first check is that there cannot be more than one OP_RETURN per transaction. The variable noopreturn is incremented by 1 each time a transaction output starts with a OP_RETURN opcode. This cannot be verified with a real transaction as they are invalid.

There is a variable DEFAULT_PERMIT_BAREMULTISIG with a value of true. This value can be changed using a parameter permitbaremultisig on the bitcoin command line. We have a transaction with an output type of multisig but baremultisig is permitted and hence no error is shown.

The option baremultisig decides whether the multisig script should be permitted in the output or not. There is a subtle difference between the multisig script being in the output and in the input. The inputs are read at the time of proving ownership of the Bitcoin, once proved we can prune away all the inputs. The outputs however are read when the ownership of the Bitcoins is transferred to someone else. At this point, the coins go from unspent to spent. Placing something in the outputs mandates all nodes to store this multisig transaction for a long time or forever till these coins are spent.

On paper at least, there is no such need to download an input as the checks can be carried out by someone else. Validation checks for the inputs are done once, when the output comes in. Not when we spend the bitcoins, obviously at a future date. The input data precedes the execution of an output script. Google disappoints big time while searching for this option, permitbaremultisig. It was first introduced in Bitcoin Core Version 0.13.

The Solver function flags a transaction as scriptpublickey on three different occasions and it also returns a false and

marks this transaction as a non-standard one. The first time is when a witness is enabled and the transaction type is not one of the two seen earlier. The second time is when the type of multisig, m and n are less than 1 and m is not greater than n. The third time is when the templates which are checking for a pubkey and pubkeyhash fail. This comes from reading the source code, so incoherent to all of us.

Any deviations from the checks makes the transactions non-standard and an error is flagged. These standard transaction tests can be ignored for the test and regtest networks.

```
Ch3702.py
import subprocess
import json
WITNESS_SCALE_FACTOR = 4
def getscripts(s, t):
    retlist = []
    if len(s) == 0:
        return 0
    if ord(s[0]) == 0:
        s = s[1:]
        i = len(s)
        j = 0
        sigcount = 0
        if t == 'pubkeyhash' or t == 'pubkey':
            while i >= 1:
                opcode = ord(s[j])
                if opcode >= 1 and opcode <= 75:
                    j = j + opcode + 1
                    i = i - opcode - 1
                else:
                    retlist.append(opcode)
                    j = j + 1
                    i = i - 1
            if 0xac in retlist or 0xad in retlist:
                sigcount = sigcount + 1
            elif t == 'inputs':
                while i >= 1:
                    opcode = ord(s[j])
                    if opcode >= 1 and opcode <= 75:
                        j = j + opcode + 1
                        i = i - opcode - 1
                    elif opcode == 0x4c:
                        offset = ord(s[j+1])
                        rest = s[j+1:]
                        if ord(rest[-1]) == 0xae:
                            sigcount = ord(rest[-2]) - 0x50
                        else:
                            sigcount = 0
                        break
```

```
        elif t == 'witness':
            if ord(s[0]) >= 0x52 and ord(s[0]) <= 0x60:
                sigcount = sigcount + ord(s[-2]) - 0x50
            else:
                sigcount = 0
            return sigcount
sighash =
['815d1b81c4ae5ad9c9cd72b89cc19b9e316477b5864bbe63a2de0b360f4dc36b',
'4acd2868bb748b0098878bce27d60e31c727b15230fb78aa99ed9884f0503879',
'6d9b32e25ac6e074f8faa38b9a261012609faa287836818b32185c7401512802',
'fb080796ef2fba32a5a3185740517929cee3763df1956b4f969fd5c6f1f7307a',
'2ce43a165be6546a2ea322449bd05c64d12bf095dc60f9aa5574bbac183aa1de',
'200afaf887885b79422cbc2587c947a1265427e0e88821f2e3fce904ad039875',
'e06615fd8b44217eade55bfcae26fe935566530210bee37896f69ec0353430a7',
'08991b46c1b26774ca00f95368c362d34fb8b3dc151cc097cb5b6363c5741daf',
'248b3e491417b4b9fccbeddb0d14d6ff9e64405e56a142b1b0b077e21453d2ea',
'1f015bd404eae7563828b46a8be56c94d90bd0c4566d443704658b9df6002380',
'e06615fd8b44217eade55bfcae26fe935566530210bee37896f69ec0353430a7',
'fa5dc10eb723c572592a2f803340febccb6c94ab009fae03a3db76e42077983b',
'3c39d930831cee652c9dbef709cf86c060e52a3a816a01588e4f452c74cec373',
'2ed5fdd426acdffb40be1f3f6b698d67538744b385f458f6a794b5e1a3016764',
'2ea458427d6573f1dceba97b45ff5bfd3e7844b9afc50e848cd02496a39c21b2',
'62ca3b5fff1204a70bf58dd09297937ff0be831a7b4c68c701c5d1565b323ef5',
'5dc34bd5db70a0805f0823da164b24a6ff6b59572c84114dff4de8aceceb2d37',
'5437b8d0fba6bd065c9ad3f7f8471e6929a943a6b818a0ede0812d768e72d28b',
]
sigops = [7 , 10 , 9 , 13, 3 , 11, 30 , 15 , 64 , 53 , 30 , 9 , 20 , 3 , 22 , 32 , 18 , 16 ]
for total in range(0, len(sighash)):
    raw = subprocess.check_output(["bitcoin-cli" , "getrawtransaction" , sighash[total] , "1" ])
    print sighash[total]
    raw = json.loads(raw)
    sigcounto = 0
    for i in raw['vout']:
        cnt = getscripts(i['scriptPubKey']['hex'].decode('hex') , i['scriptPubKey']['type'])
        sigcounto = sigcounto + cnt
    sigcounti = 0
    sigcountw = 0
    sigcounts = 0
    for i in raw['vin']:
        cnt = getscripts(i['scriptSig']['hex'].decode('hex') , 'inputs')
        sigcounti = sigcounti + cnt
    witness = ''
    try:
        witness = i['txinwitness']
    except:
        witness = ''
```



```

for k in range(0 , len(witness)):
    sigcountw = sigcountw + getscripts(witness[k].decode('hex') , 'witness' )
    if len(witness) != 0 and len(i['scriptSig']['hex'].decode('hex')) == 23:
        sigcountw = sigcountw + 1

    for i in raw['vout']:
        if i['scriptPubKey']['type'] == 'scripthash':
            for j in raw['vin']:
                k = j['scriptSig']['hex'].decode('hex')
                if len(k) > 0:
                    lastbyte = ord(k[-1])
                    secondlastbyte = ord(k[-2])
                    if lastbyte == 0xae and (secondlastbyte >= 0x52 and secondlastbyte <= 0x60):
                        raw1 = subprocess.check_output(["bitcoin-cli" , "getrawtransaction" ,
                            j['txid'] , "1"])
                        raw1 = json.loads(raw1)
                        raw2 = raw1['vout'][j['vout']]['scriptPubKey']['type']
                        if raw2 == 'scripthash':
                            sigcounts = secondlastbyte - 0x50

    legacycnt = (sigcounti + sigcounto + sigcounts ) * WITNESS_SCALE_FACTOR +
    sigcountw
    print "Sigops Input %d Output %d Sighash %d Witness %d Total %d Matches %s" %
    (sigcounti , sigcounto , sigcounts , sigcountw , legacycnt , legacycnt ==
    sigops[total] )

```

Check #20. In the program, we count the signature checks carried out in a single transaction. There must be an upper limit to refrain a hacker from conducting a Denial of Service attack. Also, it prevents miners from wasting resources on useless signature checks. There is a variable called `MAX_BLOCK_SIGOPS_COSTS` with a very large value, 80,0000. The number is the maximum limit for signature checks. These error checks serve no purpose as such.

The problem at hand is that there is no function that given a transaction comes back and reveals the sigops present in a transaction. The only way to out is to first locate a function called `GetTransactionSigOpCost`, which calculates the sigops. The transaction hash and the value returned by this function are printed. For the record, this function is called only twice. Without the source code, there would be no known way to find out the sigops cost per transaction. What complicates it further is that this sigopcost is calculated using three different methods. The basic one is called legacy and it is always calculated. Then we bring in the P2SH transaction type and finally witnesses, which are handled differently. We confess that this last program in our book works only sometimes. Too much effort to support all transactions.

All the real work is done in the `getscripts` function. Every output of the transaction is given to this function, the second parameter is for the type inputs or witness etc. The function first checks if the hex field byte has a length and the first byte is not 00. If yes, the byte is skipped. Then the output type is checked to be pubkeyhash or pubkey. We first check if the opcode is less than 0x75, if yes, we skip that many bytes. The variable `j` determines which byte to read and the variable `i` keeps track of the length of the data.

The objective here is to append all the opcodes that we have found while scanning the outputs into a list, `retlist`. For instance, if there is an opcode `OP_CHECKSIG` or `OP_CHECKSIGVERIFY`, then the `sigcount` variable is incremented by 1. The `sigcount` variable stores the number of sigops found so far.

The inputs are scanned using the same function, `getscripts` with the second parameter having a value of inputs. The code gets repeated as the same data is scanned again looking for opcodes between 0 and 0x75, like before. We search for an opcode, 0x4c or `OP_PUSHDAT1`. We extract all the bytes after pushing the signatures on the stack. Then the last byte is checked to be a `OP_CHECKMULTISIG`. We should have also checked for `OP_CHECKMULTISIGVERIFY` but we did not. Then like before, we subtract the `n` or the number of signatures by 0x50. This finally gives the number of sigops in this input. The value is stored in the `sigcounti` variable.

The witnesses are handled by checking for opcodes `OP_1` and `OP_16` using the code written for multisig types. Towards the end, we also multiply the sigops from the inputs and outputs by 4.

We have intentionally overlooked one of the most common types which is scripthash. It is the most complex hash type in the system.

In every output, we check only for the scripthash type. Then things change. All the inputs are scanned. Our first check is on the key `scriptSig`, which is normally absent for witnesses. The length of the `scriptSig` must be larger than 0. Then comes a kludge, we extract the last and the second last byte of the signature+public key. There is a sanity check to checks if the last byte is a multi sig opcode and the second last byte is `OP_??` instruction.

The output that the input transaction hash is pointing to is retrieved using the `getrawtransaction` method. We are interested in only the output that brings in the bitcoins. If the output type is scripthash, then and only then, 0x50 is subtracted from the second last byte or the `n` of `m` of `n`. This value is a count of sigops and it is returned. Why we need this check is still a mystery to me. Along the way, we could have checked if the coin was spent or not.

Yes, we should have used the `getblocktemplate` method to read the transaction hash and the sigops. The only problem is that the number of transactions are not pre-decided and they are not present in the millions. It is difficult to check sigops of certain transactions as the transactions output of `getblocktemplate` method is very dynamic. Finally, the maximum sigops count is so high that we will never find a transaction that will come even within fighting distance. So, why waste time with a few sigops here and there.

This is the last chapter looking at C/C++ and Python code. We are on a break now. Our next book is on creating digital tokens or an Initial Coin Offering or a ICO with a prime focus on security. This is how future generations will raise money. Solidity programming!!!!. Instead of having lawyers wet contracts, programmers will certify smart contracts.

CHAPTER 38

ICO and Smart Contract Security

One big use of Ethereum Smart Contracts is in the writing of ICO's or an Initial Coin Offering (ICO).

Let's go back to the history of raising funds. In the good old days, there were very few options to raise money. The first option was to take a mortgage on the house or pawn the jewelry. The second was to borrow money or take a loan from family and friends. In the unofficial sector, one's reputation and goodwill decided how much money he/she could bring in. The third option would be a bank loan, which requires some loan security, lots of paperwork and answering too many questions. The harried bankers believe that you will never repay the loan. They are very right in thinking so, as people have disappeared or left countries to avoid the repayment.

The fourth option, an IPO. A big merchant banker or Wall Street types would entice the public to invest in your company. Here the stock exchange regulator drowns you with paperwork and the suits take their pound of flesh. The biggest problem with an IPO is that it is nation based. As an Indian, it's very difficult to subscribe to an IPO in the US or Europe. Too many laws, rules and regulations.

The Internet bought in a fifth option which was crowdfunding or Kickstarter. If you had a great product that is better than sliced bread or a blockbuster movie, ask the world to fund or crowdfund it.

We needed a better way. Enter digital tokens. The idea is brilliant. You value your idea at say a million Bitcoins. A smart contract is written and people are asked to pay in Ether or Bitcoin. A new approach to raising capital. This idea funded the development of Ethereum, the idea of an ICO or a digital token. The people who bought ether for Bitcoin are very rich people today.

Today one of the best ways of making money is by coming out with a ICO. All that is required of you is an impressive power point presentation. Zero paperwork. If people want to, they will buy your digital tokens for Bitcoins or ether or even dollars.

Is an ICO legal? No one knows as every country has its own set of laws and regulations. In India, nobody knows whether buying a Bitcoin is legal or not as the regulator is silent on this issue. The biggest problem with a ICO is that it removes the state, the stock exchange and monetary regulator and Wall Street out of the picture. That's why there is so much opposition to an ICO. The biggest advantage of a ICO is that it is digital, one can invest as low as millionth of a cent or for that matter, could be sitting on Mars for all you care.

You are not diluting a stake in your company when you use an ICO to raise funds.

For some years, an ICO will be a lawyer's paradise. It is our belief that countries will gradually be drawn towards it, when they witness huge funding to the countries who have legalized an ICO as a legitimate means of raising money. Thus, over time, all countries will allow money to flow into an ICO, too big to be ignored. We are not lawyers to debate whether an ICO should be treated as an asset or a security or whatever. We leave that to the high and mighty, our job is to create an ICO now.

You cannot write an ICO in Bitcoin because the opcodes of Bitcoins do not support a programming language. Ethereum was designed to be programmable as its Turing complete. Let's move from BIP or Bitcoin Improvement Proposal to a

EIP or Ethereum Improvement Proposal or EIP-20. This is a Ethereum standard that defines a digital token or simply a token. This EIP carries a lot of weight because the Ethereum God, Vitalik Buterin and Fabian Vogelsteller have co-authored it themselves. Therefore, one ICO to another will look and feel the same. We have standardization and a roadmap for the future.

We intend to get our smallest contract, Mukhi working despite the fact, that our contract cannot be compiled anymore in geth. We keep wondering why the compile function was disabled in geth and why was no one informed about it. Nobody seems to have gained, we are all losers. Many people have uploaded lots of workarounds, the approach taken here is what we felt was the best way of creating contracts.

In a terminal write the following.

```
y
Vijay Mukhi
>a="cat y"
```

We are creating a bash shell variable called a, whose value is a command but seen as a simple text string.

```
>echo $a
cat y
```

To display the value of a shell variable, the echo command is used with a dollar sign in front of the variable.

```
>a='cat y'
```

We now replace the double quotes by the reverse single quotes below the tilde key. The tilde quotes indicate running the program, so the contents of the file called y are displayed and the value is played in the shell variable a. Make sure you have a dummy file y with some text in it, we have my name in it.

```
>echo $a
Vijay Mukhi
>echo "var myname=`solc --combined-json abi,bin vijay.sol`" > sonal.js
```

This is kindergarten stuff for any Unix/Linux administrator. But lots of people do not use the shell terminal window.

```
vijay.sol
pragma solidity ^0.4.17;
contract Mukhi {
    function vijaymukhi() constant public returns (string)
    {
        return "Vijay Mukhi is a embecile";
    }
}
```

This is the smallest contract that we can run. The pragma statement informs the solidity compiler about the solidity version used in the program.

```
>solc --bin vijay.sol
606060405234156100
```

Earlier, the solidity compiler with the --bin option displayed a series of bytes, we asked you to copy and paste these bytes in geth in an earlier chapter.

Let's try another option of the solc compiler, combined-json.

```
>solc --combined-json bin vijay.sol
{"contracts":{"vijay.sol:Mukhi":{"bin":"6060604052341
```

Notice the difference, now we have a json output or a Javascript dictionary. The key is called contracts, which is a dictionary object. It has one key called vijay.sol, the name of the program and then the name of the contract Mukhi. This key is also a dictionary, it has another key called bin followed by the actual bytes. This is great but the output must be a Javascript variable. So, modify the code to the following:

```
>echo "var myname=`solc --combined-json abi,bin vijay.sol`"
var myname={"contracts":{"vijay.sol:Mukhi":{"abi"
```

Now we are getting someplace. A variable called myname is created and its value is the output of the solc command. The solc command takes one option abi as we require the bytecodes and method signatures. But now it must be in a file.

```
>echo "var myname=`solc --combined-json abi,bin vijay.sol`" > sonal.js
```

This command simply directs the standard output to a file, sonal.js. Now this Javascript file can be loaded in geth.

This is how the shell script y looks like

```
rm sonal.js
echo "var myname=`solc --combined-json abi,bin vijay.sol`" > sonal.js
a.js
loadScript("sonal.js");
vijayabi = JSON.parse(myname.contracts["vijay.sol:Mukhi"].abi)
vijaybin = "0x" + myname.contracts["vijay.sol:Mukhi"].bin
console.log(vijaybin)
>loadScript("a.js")
0x6060604052341561000f5760
```

We create a file called a.js which loads the Javascript file sonal.js which was created as we ran the program y first in a separate terminal and then moved into geth and loaded the file a.js.

When the loadScript function gets executed in file a.js, a variable called myname of type dictionary gets created. The contracts key is extracted and then using the array notation, the key vijay.sol:Mukhi is used to gain access to the two other keys, bin and abi. The variables vijayabi and vijaybin are used to access the same data obtained on compiling our code using the geth compile function.

To sum up, we run the solc compiler outside of geth using the shell script y and then use loadScript within geth to run the script a.js. This way we undo not having been able to compile within geth.

```
a.js
loadScript("sonal.js");
vijayabi = JSON.parse(myname.contracts["vijay.sol:Mukhi"].abi)
vijaybin = "0x" + myname.contracts["vijay.sol:Mukhi"].bin
var c = eth.contract(vijayabi);
var v = c.new({from:eth.accounts[0], data: vijaybin , gas:1000000} , function (e , c) {
  console.log("e " + e + " c " + c.transactionHash + ":" + c.address);
});
```

Now that we have the abi and bin, we use code from our earlier chapter and this is the output.

```
loadScript("a.js")
e null c
0xdb95c570d8896d20730fb139ea676573ddec0af7eb0300aa0da88a98ccbacf84:undefined
true

> e null c
0xdb95c570d8896d20730fb139ea676573ddec0af7eb0300aa0da88a98ccbacf84:0x8cf64ffba
7f1aa09772ea4e52adcc42c130c5f1b

> v.vijaymukhi()
"Vijay Mukhi is a embecile"
```

Everything now works as advertised.

We repeat ourselves for the last and final time.

```
vijay.sol
pragma solidity ^0.4.17;
contract ERC20{
    string public constant symbol = "VijayMukhi";
    string public constant name = "An example of a Useless Token";
    uint8 public constant decimals = 18;
}

echo "var myname=`solc --combined-json abi,bin vijay.sol`" > sonal.js
```

This creates the Javascript file, sonal.js, which gives access to abi and bytecodes.

```
a.js
loadScript("sonal.js");
vijayabi = JSON.parse(myname.contracts["vijay.sol:ERC20"].abi)
vijaybin = "0x" + myname.contracts["vijay.sol:ERC20"].bin
var c = eth.contract(vijayabi);
var v = c.new({from:eth.accounts[0], data: vijaybin , gas:1000000} , function (e , c) {
    console.log("e " + e + " c " + c.transactionHash + ":" + c.address);
});
```

The class having a digital token code is now called ERC20 and not Mukhi and the contract in the a.js file is also called ERC20. This name is a standard that all examples use. So, we stick to the standards and rename our contract to ERC20 in our file, Vijay.sol. There are three public variables created in the contract. Anyone can access them but not change them or write to them, so it's a bad idea to make them public. These variables are optional but it's a good idea to follow the ERC-20 standard to the T. The symbol is the currency symbol of our new cryptocurrency. Since the dollar and euro were already taken, my name, VijayMukhi turned out to be the next best choice.

The variable called name has a description of the token. This value assigned to this variable is not important for now. Finally, the number of decimals is 18, taken straight from the standard. The maximum account balance you can have, is a number that ranges from 0 to 115 quattuorvigintillions. The official tutorial defines a quattuorvigintillion as lots and lots of vigintillions, but it does not define a vigintillions

The idea is that we can create any number of tokens but it must be multiplied by the number given in decimals.

We get the following output.

```
loadScript("a.js")
e null c 0x96dd96c82b5c357cf395b8b78dbe2b3bd4edce199af300b6baa393bca4538c2d:undefined
true
> e null c
0x96dd96c82b5c357cf395b8b78dbe2b3bd4edce199af300b6baa393bca4538c2d:0x0e437b07
cfd6e47e62a284ddd94054483afdca22
> v.symbol
function()
> v.symbol()
"VijayMukhi"
> v.name()
"An example of a Useless Token"
> v.decimals()
18
```

```
vijay.sol
pragma solidity ^0.4.17;
contract ERC20 {
    address owner;
    uint256 dummy;
    function ERC20(uint256 _useless) public {
        dummy = _useless;
        owner = msg.sender;
    }
    function a1() constant returns (uint256) {
        return dummy;
    }
    function a2() constant returns (address) {
        return owner;
    }
    function a3() constant {
        dummy = 1000;
    }
    function b1() returns (uint256) {
        return dummy;
    }
    function b2() returns (address) {
        return owner;
    }
    function b3() {
        dummy = 1000;
    }
}
```

The file vijay.sol now has lots of function that have nothing to do with the ERC standard. We start with a function name that is the same as the Contract name or the class name, ERC20. These functions are special, they are called constructors. A constructor can never be called externally, they get called once when the contract is created. In our case, the ERC20 constructor gets called with a single integer called `_useless`.

This is how our revised a.js file looks.

```
a.js
loadScript("sonal.js");
vijayabi = JSON.parse(myname.contracts["vijay.sol:ERC20"].abi)
vijaybin = "0x" + myname.contracts["vijay.sol:ERC20"].bin
var c = eth.contract(vijayabi);
var v = c.new(420 , {from:eth.accounts[0], data: vijaybin , gas:1000000} , function (e , c) {
    console.log("e " + e + " c " + c.transactionHash + ":" + c.address);
});
```

One change, the new function off the contract is called with some extra values. The first parameter has the arguments to be passed to the constructor. In our case, we have 1 parameter. The `_useless` variable in the constructor will now have a value of 420.

The class has two instance variables, dummy and owner. They are set to the value passed to the constructor, 420 and the address of the account that is funding this contract.

The free object msg has one member called sender. In our case, its value is `eth.accounts[0]`, the one who is funding everything.

```
> eth.getBalance(eth.accounts[0])
2886898090000000000
```

Each time you run a `geth` command, please check the Balance in the first member of account. This way we will understand when the ethers are getting spent to fund our transaction. The value is in wei, an Ethereum currency unit.

```
> loadScript("a.js")
```

We run a.js. There is no wait in creating the transaction but there is a wait for up to 5 minutes to get a valid address.

```
e null c
0x1c957b860b24956ab1bd687f305418f825b2d26ed73566dcc810a7aee087bd32:0xacff68fa4
f175b28704a0bf4c140512965152603
```

Once we get this output, we are all set to go. We first find out the balance we have left.

```
> eth.getBalance(eth.accounts[0])
2881792310000000000
```

Nothing is free, our account balance reduces by a 5105780000000000. The blockchain explorer gives the same value but in ether. We are doing something right.

```
> v.a1()
420
```

In our contract, we run our first function called `a1`. The function returns the value of the dummy variable, which is 420. The value was set in the constructor. Let's now run the `a2` function.


```
> v.a2()
"0xbf254211c6b0a6a2882e269008b0024f3e6b5270"
> eth.accounts[0]
"0xbf254211c6b0a6a2882e269008b0024f3e6b5270"
```

As we said earlier the free object msg has a member called sender whose Ethereum address is the same as the address of our first account. Basically, the account funding our contract.

```
> v.a3()
[]
```

We run the third function called a3 which simply changes the dummy variable to 1000. Now we run the a1 function to check if its value is 1000. Plus, we ask for the balance.

```
> v.a1()
420
> eth.getBalance(eth.accounts[0])
2881792310000000000
```

The output knocked us out. First, the a1 variable shows the same value 420 for dummy even though we changed it to 1000. And secondly, the account balance does not change a wei. This is sarcasm at its best.

You need working code to understand the fundamental principles of Ethereum's world computer. Our code is running on millions of miners all over the world. The value of variables owner and dummy are cast in stone in the blockchain. These values are initialized in the constructor and they cannot be changed. One of the programs in an earlier chapter explained the complexity of the state machine. But, if we cannot change instance variables then what good is Ethereum for us.

The principle is that you can change but in a different way. The keyword constant implies no change to the instance variables. The solidity compiler does not complain if you do so, but will not act on it. A constant function should only read and not write or change instance variables. Local variables are second class citizens as they die at the end of the function any which way.

Let's now run the b1 function which does not have the constant keyword. It must return the value of the dummy variable.

```
> v.b1()
Error: invalid address
```

We are simply not allowed to run a function having no constant keyword. An Invalid Address error suggests that this function is not present in our contract. There is a only one way to change state and that is through a transaction.

```
> v.b3.sendTransaction({from:eth.accounts[0]})
"0x3d62740136d4a39530ba15e04fc0d8fa6f31fdc1dba4b9c1750a6e6e468dc8a0"
```

Now we call a function, sendTransaction off the b3 function. The b3 function does not have the constant keyword. We are returned a transaction hash that can make the required changes to the dummy variable.

```
> eth.getBalance(eth.accounts[0])
2881792310000000000
```

However, the balance function shows no gas used. We waited for some time and this is the new balance.

```
>eth.getBalance(eth.accounts[0])
2881262330000000000
```

The gas used now is 52998000000000, reconfirmed by our blockchain explorer.

The icing on the cake for us was this command.

```
>v.a1()  
1000
```

To sum up, the keyword constant does not allow any changes to state variables. If you want to make changes, remove the constant keyword but then create a new transaction for which you have to pay gas and then wait for the transaction to get confirmed.

We love doing these things.

```
> v.b2.sendTransaction({from:eth.accounts[0]})  
"0x11417d31542dd1f566f8e081151bbf5e4227e1c8c0badcd7e0810cdbb0b1ace1"
```

We now execute the b2 function which does not change any state. A new transaction get created.

```
>eth.getBalance(eth.accounts[0])  
2880828110000000000
```

We are using up ether even though we made no change to the state. Figure out the differences between the two gas you paid.

```
vijay.sol  
pragma solidity ^0.4.17;  
contract ERC20 {  
    mapping (address => uint256) public balances;  
    function ERC20 (uint256 startvalue) public {  
        balances[msg.sender] = startvalue;  
        balances[0x1111111111111111111111111111111111111111111111111111111111111111] = 20;  
        balances[0x1111111111111111111111111111111111111111111111111111111111111112] = 30;  
    }  
    function transfer ( address _to , uint256 _value) public returns (bool success){  
        require(balances[msg.sender] >= _value);  
        require(balances[_to] + _value >= balances[_to]);  
        balances[msg.sender] -= _value;  
        balances[_to] += _value;  
        success = true;  
    }  
    function balanceOf ( address _owner ) constant public returns (uint256 balance) {  
        balance = balances[_owner];  
    }  
}
```

The program now uses the most important data type, which is a complex array or an associative array, something missing in languages like C. We need a data structure that will store an Ethereum address, like our bank account number or ICO account address and the balance associated with that account.

A simple dictionary is perfect for it but it requires a lot of housekeeping. Now here comes the mapping keyword. We specify the data type of the key, which is an address in our case and the data type of the value associated with this key.

This variable of type mapping, for want of better word, is called balances and it is made public for no coherent reason. Instance variables must be private in our view. In the contract ERC20 constructor, we create three keys in this balances variable. The first key is for our Ethereum address where we store a count, the number of coins that make up our ICO. We use the [] brackets for the key followed by the equal sign and this key's value. Solidity does the rest for us.

A Ethereum address is nothing but a 20-byte hash value. Two more keys are created ending with a digit 1 and 2 and the values are set to 20 and 30. Then a function called balanceOf is created which gives access to the value of any key passed as a parameter. This function is of type constant since no changes are to be made to our state.

We then execute our script and run the following commands.

```
> v.balanceOf("0x1111111111111111111111111111111111111111111111111111111111111111")
20
> v.balanceOf("0x1111111111111111111111111111111111111111111111111111111111111112")
30
> v.balanceOf(eth.accounts[0])
420
> v.balanceOf("0xbf254211c6b0a6a2882e269008b0024f3e6b5270")
420
```

The balanceOf function is given an Ethereum address but in a string format. The string type is a must or else the value gets converted into a hex number and then is used as a key. Our first Ethereum account and the sender field in the msg object must have the same value.

The Ethereum address once again must be placed in double quotes. We wasted too much time because we initially did not.

Now comes the clincher.

```
>v.transfer.sendTransaction("0x1111111111111111111111111111111111111111111111111111111111111112", 10 ,
{from:eth.accounts[0]})
"0x59d7ed537e51822973f91bffe8bed7e9f293fe243ffc74c3d09a59dd31c0cf1"
```

This transaction hash shows success. As this function transfer does not use the constant keyword, it must be called using the sendTransaction function. The transfer function is very simple, it has only two parameters, the first is the Ethereum address we want to transfer our cryptocurrency to and the second is the amount of money/ethers.

The code is very simple, the balance in the Ethereum account denoted by msg.sender or by eth.accounts[0] is reduced by the amount to be transferred. It is a must. The transfer value is then passed as a parameter. It is increased by the same value in the receiver's account.

Error checks are a must when you transfer money. The first condition is that the owner has money which is larger than _value to transfer. All parameter names in solidity must start with a _. This is God's law. This condition is placed in a require, based on the result, it can pass and return success or it will halt the execution of the contract, also undo the state and return Failure in the browser explorer. Replace the >= sign in the first require with a < and see the error yourself and that the state did not change a wee bit.

The second error check is to make sure that the to account do not encounter an overflow. The too balance and the value to be added must be larger or equal to the too balance. When numbers grow too big, they get negative.

In solidity, whenever a variable name and its datatype are returned, a variable name is assigned the return value.

```
>v.balanceOf("0x1111111111111111111111111111111111111111111111111111111111111112")
40
```

The balance of account ending with 2 was 30 and its now 40.

The blockchain explorer displays the following in the data section. Interesting.

```
Function: transfer(address _to, uint256 _value) ***
MethodID: 0xa9059cbb
[0]:0000000000000000000000000000000000000000000000000000000000000000
[1]:000000000000000000000000000000000000000000000000000000000000000a
```

The transfer function is called with two parameters. The first parameter to the transfer function is an address ending with 2 and the second one is a value, 10 or a in hex. The two fields are 64 digits long.

We have looked at two functions from the ERC20 standard, balanceOf and transfer, four more to go.

```
vijay.sol
pragma solidity ^0.4.17;
contract ERC20 {
    mapping (address => uint256) public balances;
    uint256 totalofcoins;
    function ERC20 (uint256 startvalue) public {
        balances[msg.sender] = startvalue;
        totalofcoins = startvalue * 10 ** 18;
        balances[0x1111111111111111111111111111111111111111111111111111111111111111] = 20;
        balances[0x1111111111111111111111111111111111111111111111111111111111111112] = 30;
    }
    function transfer ( address _to , uint256 _value) public returns (bool success){
        require(balances[msg.sender] >= _value);
        require(balances[_to] + _value >= balances[_to]);
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        success = true;
    }
    event Transfer(address indexed _from, address indexed _to, uint256 _value);
    function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
        balances[_from] -= _value;
        balances[_to] += _value;
        Transfer(_from , _to , _value);
        success = true;
    }
    function balanceOf ( address _owner ) constant public returns (uint256 balance) {
        balance = balances[_owner];
    }
    function totalSupply () constant public returns (uint256 totalsupply) {
        totalsupply = totalofcoins;
    }
}

>v.totalSupply()
4200000000000000000000000000000000
```



```
b.js
var event = v.Transfer( {}, function(e, r) {
    console.log("In Event: Addresses From " + r.args._from + " To " + r.args._to + " value " +
        r.args._value);
});
var thash = v.transferFrom.sendTransaction("0x1111111111111111111111111111111111111111111111111111111111111111",
    "0x11111111111111111111111111111111111111111111111111111111111111112", 3, {from:eth.accounts[0]} );
console.log(thash)

e1 null c
0x8fd23bcaa9dcd2b02a376fb6efd24f58eab4cb36657a7e73d8f731007163892e:0xd3b12a9b2
a861d9e0e6e62d23cb800e566730cff
```

The output after running a.js is displayed. Then loading b.js shows the following:

```
> loadScript("b.js")
0x89ec550f3f593f0fc5c9a119dd64b8b69f36170784e5e45f5ee2a319cf82871f
true

> In Event: Addresses From 0x1111111111111111111111111111111111111111111111111111111111111111 To
0x11111111111111111111111111111111111111111111111111111111111111112 value 3
```

The transaction id returned after running the sendTransaction command carries the event data with it.

We create an event object by accessing the Transfer event from our runtime object, v. The callback function is called each time our event gets fired. Our event is triggered in the transferFrom function. The args field contains the 3 parameters to our event.

Now our client code gets called when an event gets triggered, in this case, money transfer from one Ethereum address to another. A wallet or any app must monitor events in real time and get informed asynchronously. The event variable is displayed and the topics remain the same.

We spent over an hour looking for a simple example to demonstrate an event. Most of the code we found does not work. Working with testnet has been our biggest bugbear. Sometimes it works well and sometimes the wait is for hours because testnet refuses to sync.

```
vijay.sol
pragma solidity ^0.4.17;
contract ERC20 {
    mapping (address => uint256) public balances;
    uint256 totalofcoins;
    mapping (address => mapping (address => uint256)) public allowed;
    function ERC20 (uint256 startvalue) public {
        balances[msg.sender] = startvalue;
        totalofcoins = startvalue * 10 ** 18;
        balances[0x1111111111111111111111111111111111111111111111111111111111111111] = 20;
        balances[0x1111111111111111111111111111111111111111111111111111111111111112] = 30;
    }
    function transfer ( address _to , uint256 _value) public returns (bool success){
        require(balances[msg.sender] >= _value);
        require(balances[_to] + _value >= balances[_to]);
    }
}
```

```

        balances[msg.sender] -= _value;
        balances[_to] += _value;
        success = true;
    }
    event Transfer(address _from, address _to, uint256 _value);
    function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
        uint256 allowance = allowed[msg.sender][_from];
        require(balances[_from] >= _value && allowance >= _value);
        balances[_from] -= _value;
        balances[_to] += _value;
        Transfer(_from , _to , _value);
        allowed[msg.sender][_from] -= _value;
        success = true;
    }
    function balanceOf ( address _owner ) constant public returns (uint256 balance) {
        balance = balances[_owner];
    }
    function totalSupply () constant public returns (uint256 totalsupply) {
        totalsupply = totalofcoins;
    }
    event Approval(address indexed owner, address indexed spender, uint256 value);
    function approve(address _spender, uint256 _value) public returns (bool) {
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        return true;
    }
    function allowance(address _owner, address _spender) public constant returns (uint256 remaining) {
        return allowed[_owner][_spender];
    }
    address public aa;
    function zzz () {
        aa = msg.sender;
    }
}

```

Technically this is the last program that will make us ERC20 compatible. Let's start at the bottom of the barrel or code. We have a non-constant function called zzz that takes a public variable called aa of type address and sets its value to the msg.sender's value.

```

loadScript("a.js")
e1 null c
0x94af2b2368185b935a5d92c5f370180a266c7a16fae08bce079c0f4182a590ad:0xa00ecb73e
17de63c19260755d22f9d3bd42129bc
v.ddd.sendTransaction({from:eth.accounts[0]})
"0x4d96d8406c0f53bc66bd930941ddb9f76893ae328bc20b6942f6ad96fa5081d5"

```

```
> v.aa()  
"0xbf254211c6b0a6a2882e269008b0024f3e6b5270"
```

When we call the zzz function with the first Ethereum account address, the value of msg.sender will be this Ethereum address starting with bf25. The value of aa confirms this.

```
> v.zzz.sendTransaction({from:eth.accounts[1]})  
"0x43f1184456863c961cf742dd59c587096d0a5b10e171b98565aa528386dd7d18"  
> v.aa()  
"0x08f3556f206e352303857e1567c307b06a3a9e22"  
> eth.accounts  
["0xbf254211c6b0a6a2882e269008b0024f3e6b5270",  
"0x08f3556f206e352303857e1567c307b06a3a9e22"]
```

The only change incorporated is the use of the second Ethereum address, eth.accounts[1] as the sender or owner of this transaction. The value of variable aa reflects this new Ethereum address.

A non-constant function for want of a better word, will have an extra parameter, which is the funder or caller of the transaction passed in the from key. We will keep referring to the first or second Ethereum address, it will differ in your case.

```
v.approve.sendTransaction("0x11111111111111111111111111111111", 5 ,  
{from:eth.accounts[0]})  
"0xabc8fc0065885c0f0a03b78c5ab94545f4cdadae432cda57078b20f8b2ee32de"  
v.allowance(eth.accounts[0] , "0x11111111111111111111111111111111")  
5  
v.allowance(eth.accounts[0] , "0x11111111111111111111111111111112")  
0
```

The approve method is executed first. This rationale behind having this method is that the caller of this transaction, the first Ethereum address gives permission to the Ethereum address ending with 1 to transfer up to 5 coins.

The underlying assumption is that the Ethereum address ending with 1 can transfer or approve of 5 coins. In the approve function, a map is taken of 2 keys both of which are addresses. This lets us map the Ethereum address which gave permission to the other Ethereum address to transfer coins. The first level or key will be the owner of the transaction and the second the person who we are giving approval to.

The allowance function specifies which owner gave which Ethereum address how much allowance or permission. In our case, Ethereum account 0 and Ethereum address ending with 1 has an allowance of 5. On the other hand, Ethereum address 1 and ending with 2 has no allowance. Let's change that.

```
>v.approve.sendTransaction("0x11111111111111111111111111111111", 5 ,  
{from:eth.accounts[1]})  
"0xb3fffd44223c6ac9beba4318e5f29e22faae0b9bdd88eba35760c6de6bcb9d9f"  
>v.approve.sendTransaction("0x11111111111111111111111111111111", 5 ,  
{from:eth.accounts[1]})  
"0xa90c3a226628328e4f5f80c612c43cf3e6051fb5cab5fe37b08fc290c5033ad8"  
>v.approve.sendTransaction("0x11111111111111111111111111111111", 5 ,  
{from:eth.accounts[1]})  
"0x3c7218e7600a8ecd33ae81adcb998ba7c1a700593f9f16c093838ebb45c62a4c"v.allowance  
(eth.accounts[0] , "0x11111111111111111111111111111112")
```


We go overboard and send it three approvals. The net result is that Ethereum address 1 and Ethereum address ending with 2 have an allowance of 5 coins only. Only one approval is useful.

Here we make a normal transfer of two coins between Ethereum address ending with 1 and 2 using our `transfer` function. We see that balances have reduced by 2 to 18 in the sender and increased by 2 to 32 in the receiver. This only proves that this function works.

This is the clincher. The double map allowed now has an allowance of only 3 coins as it has used up 2 coins in the last balanceOf. The allowed map is accessed directly and not through the allowance function. For the record, we do not use public instance variables.

Finally, we try to send 12 coins. The only problem is that Ethereum address ending with 1 does not have the permission to send 12 coins to anyone. That's why this transaction in a blockchain explorer says Fail in red. You need permission to transfer your own coins. Just one way of bringing in permissions.

Like before, we now trap the Approval event. No major surgery to the b1.js program. We see the Ethereum address and the value and the owner or eth.accounts[0] which triggered this event.

All our Solidity code gets converted into the Ethereum Virtual Machine bytecodes. These bytes codes are visible in any Ethereum blockchain browser, they are present in the blockchain. Let's write a disassembler that reverse engineers the bytecodes back to the original source code.

```
ch3801.py
import json
f = open("bin.data")
all = f.read()
b = json.loads(all)
b = b['contracts']['b.sol:Test']['bin']
print b
i = 0
inst = {
    0x0: ["STOP", 0 ],
    0x8: ["ADDMOD", 4 ],
    0x15: ["ISZERO", 0 ],
    0x29: ["", 0 ],
    0x20: ["KEECACK256", 0 ],
    0x27: ["0x27", 0 ],
    0x34: ["CALLVALUE", 0 ],
    0x39: ["CODECOPY", 0 ],
    0x42: ["TIMESTAMP", 5 ],
    0x52: ["MSTORE", 0 ],
    0x57: ["JUMPI", 0 ],
    0x60: ["PUSH1", 1 ],
    0x80: ["DUP1", 0 ],
    0x9d: ["SWAP14", 1 ],
    0xa1: ["LOG1", 0 ],
    0x5b: ["JUMPDEST", 0 ],
    0xf3: ["RETURN", 0 ],
    0xfd: ["REVERT", 0 ],
}
while (i < len(b)):
    byte = b[i] + b[i+1]
    byte = int(byte, 16)
    cnt = inst[byte][1]
    byte1 = ""
    while cnt > 0:
        byte1 = byte1 + b[i + 2 + (cnt - 1) * 2] + b[i + 3 + (cnt - 1) * 2]
        cnt = cnt - 1
    print "(%d) %s" % (i, inst[byte][0] + " " + byte1),
    i = i + inst[byte][1] * 2
    i = i + 2
print
```

Output

60606040523415600e57600080fd5b603580601b6000396000f3006060604052600080fd00a16

```
5627a7a7230582057ab4937000612c5dc66eac27cc730a94ab018f808404429b26cb9709b773c
4e0029
```

```
(0) PUSH1 60 (4) PUSH1 40 (8) MSTORE (10) CALLVALUE (12) ISZERO (14) PUSH1 0e (18)
JUMPI (20) PUSH1 00 (24) DUP1 (26) REVERT (28) JUMPDEST (30) PUSH1 35 (34) DUP1
(36) PUSH1 1b (40) PUSH1 00 (44) CODECOPY (46) PUSH1 00 (50) RETURN (52) STOP
(54) PUSH1 60 (58) PUSH1 40 (62) MSTORE (64) PUSH1 00 (68) DUP1 (70) REVERT (72) STOP
(74) LOG1
```

But we are getting ahead of ourselves.

```
b.sol
contract Test {}
```

You cannot get a smaller solidity file than this one.

Run the solc compiler as

```
>solc --combined-json bin b.sol > bin.data
>cat bin.data
{"contracts":{"b.sol:Test":{"bin":"60606040523415600e57600080fd5b603580601b600039600
0f3006060604052600080fd00a165627a7a7230582057ab4937000612c5dc66eac27cc730a94a
b018f808404429b26cb9709b773c4e0029"}},"version":"0.4.18+commit.9cf6e910.Darwin.
appleclang"}}
```

We ask Solidity to compile the file b.sol and give the binary output as a json output. This output is redirected to a file called bin.data. The bin field has our compiled Ethereum JVM bytecodes. These bytes are found on the blockchain. We are going to disassemble these bytes.

The Python program reads the file bin.data and then extract the field called Contracts, the field b.sol:Test which is the name of the File and the contract name.

Now comes the easy part. The structure of the bytecodes is very simple. Two characters are read at a time in the while loop. These two digits are then concatenated to give a hex number using the int function.

Every JVM instruction is known by a number. The opcode is called PUSH1 but its number is 0x60. The dictionary inst uses the opcode value as the key and the value is a list. The first member of this list is the opcode name, we display this name. The second is the number of parameters this opcode uses.

The PUSH family of opcodes pushes values on the stack. Most of the other opcodes are not passed parameters, but we read values off the stack. The PUSH family of opcodes pushes multiple values on the stack. The PUSH2 opcodes pushes two bytes on the stack. We know this because we read the Solidity source code written in an easier C++ than the one used to write Bitcoin. We have no idea which PUSH family instruction should be used. So, we also store the number of bytes pushed as the second member of the list. A while loop comes in depending upon the number of bytes to be pushed on the stack. This string is stored in the byte1 variable.

In the end, we simply display the opcode and the bytes it pushed on the stack. The variable i is increased by 2 plus the number of bytes it pushed on the stack. This ensures that we are at the start at the next opcode. The 0x60 following the opcode 0x60 is data to be pushed on the stack and not the opcode push.

```
>solc --opcodes b.sol
===== b.sol:Test =====
```

Opcodes:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH1 0xe JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST PUSH1 0x35 DUP1 PUSH1 0x1B PUSH1 0x0 CODECOPY PUSH1 0x0
RETURN STOP PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x0 DUP1 REVERT STOP LOG1
PUSH6 0x627A7A723058 KECCAK256 JUMPI 0xab 0x49 CALLDATACOPY STOP MOD SLT 0xc5
0xdc PUSH7 0xEAC27CC730A94A 0xb0 XOR 0xf8 ADDMOD BLOCKHASH DIFFICULTY 0x29
0xb2 PUSH13 0xB9709B773C4E0029000000000000
```

Our workings kind of match with the `--opcodes` option of the solc compiler. The reason we are saying kind of is because after the second stop opcode, the solc output does not make sense.

```
ch3802.py
import json
f = open("bin.data")
all = f.read()
b = json.loads(all)
b = b['contracts']['b.sol:Test']['bin']
print b
i = 0
inst = {
    0x0: ["STOP", 0 ],
    0x8: ["ADDMOD", 4 ],
    0x15: ["ISZERO", 0 ],
    0x29: ["", 0 ],
    0x20: ["KEECACK256", 0 ],
    0x27: ["0x27", 0 ],
    0x34: ["CALLVALUE", 0 ],
    0x39: ["CODECOPY", 0 ],
    0x42: ["TIMESTAMP", 5 ],
    0x52: ["MSTORE", 0 ],
    0x57: ["JUMPI", 0 ],
    0x60: ["PUSH1", 1 ],
    0x62: ["PUSH3", 3 + 2 ],
    0x65: ["PUSH6", 6 ],
    0x69: ["PUSH10", 10 ],
    0x80: ["DUP1", 0 ],
    0x9d: ["SWAP14", 1 ],
    0xa1: ["LOG1", 0 ],
    0x5b: ["JUMPDEST", 0 ],
    0xec: ["0xec", 0 ],
    0xf3: ["RETURN", 0 ],
    0xfd: ["REVERT", 0 ],
}
stop = 0
while (i < len(b)):
    byte = b[i] + b[i+1]
    byte = int(byte, 16)
```

```

cnt = inst[byte][1]
byte1 = ''
if byte == 0x00:
    stop = stop + 1
while cnt > 0:
    byte1 = byte1 + b[i + 2 + (cnt - 1) * 2] + b[i + 3 + (cnt - 1) * 2]
    cnt = cnt - 1
#print("(%d) %s" % (i, inst[byte][0] + " " + byte1)
i = i + inst[byte][1] * 2
i = i + 2
if stop == 2:
    break
print("\t\t\t\t\t %s" % b[i:])
>solc --bin b.sol
>python ch3802.py

```

We run the following two commands and this is the output we get.

```

Binary:
60606040523415600e57600080fd5b603580601b6000396000f3006060604052600080fd00a16
5627a7a7230582057ab4937000612c5dc66eac27cc730a94ab018f808404429b26cb9709b773c
4e0029
60606040523415600e57600080fd5b603580601b6000396000f3006060604052600080fd00a16
5627a7a7230582057ab4937000612c5dc66eac27cc730a94ab018f808404429b26cb9709b773c
4e0029
a165627a7a7230582057ab4937000612c5dc66eac27cc730a94ab018f808404429b26cb9709b7
73c4e0029

```

This output looks better on our computer screen. After the second stop opcode 0x00, the numbers hex 0xa1 and 0x65 are seen. These bytes are part of the comments in the Solidity source code.

```

// CBOR-encoding of the key "bzzr0"
bytes{0x65, 'b', 'z', 'z', 'r', '0'}+
// CBOR-encoding of the hash
bytes{0x58, 0x20} + dev::swarmHash(metadata).asBytes();
0xa1 0x65 'b' 'z' 'z' 'r' '0' 0x58 0x20 <32 bytes swarm hash> 0x00 0x29

```

The '0' is ASCII 48 or 0x30. The 0x58 is added for more good luck, you have just been introduced to one more encoding CBOR. The 0x20 is the length of swarm hash which in our case is.

```
57ab4937000612c5dc66eac27cc730a94ab018f808404429b26cb9709b773c4e
```

Finally, we end with the length which is 41 bytes. This is the length from the byte 0xA1 to the end of the hash, not counting the length bytes. In another book, we will explain how to compute the swarm hash. As a bonus, we will calculate the Keccak-256 hash or the SHA3 hash.

```

Keccak.py
from Crypto.Hash import keccak

```

```
keccak_hash = keccak.new(digest_bits=256)
keccak_hash.update('contract Test {}')
print keccak_hash.hexdigest()
```

Output

```
5fd67e830b7501e09568547ab6a0b33e60e07b8624f087e7e585b96f6f18d925
```

```
>solc --metadata b.sol
```

```
===== b.sol:Test =====
```

Metadata:

```
{“compiler”:{“version”:"0.4.18+commit.9cf6e910"},“language”:"Solidity",“output”:{“abi”:[],
“devdoc”:{“methods”:{}},“userdoc”:{“methods”:{}},“settings”:{“compilationTarget”:{“b.sol”:"
Test"},“libraries”:{},“optimizer”:{“enabled”:false,“runs”:200},“remappings”:[],“sources”:{“b.sol”:
{“keccak256”:"0x5fd67e830b7501e09568547ab6a0b33e60e07b8624f087e7e585b96f6f18
d925",“urls”:[“bzzr://aa33a95391a5a439a874238060da3029dba38b2789ac88511fa7b4e0332
a7424”]}},“version”:1}}
```

The metadata option generates the keccak hash of our contract.

```
ch3803.py
import json
f = open("bin.data")
all = f.read()
b = json.loads(all)
b = b['contracts']['b.sol:Test']['bin']
i = 0
inst = {
    0x0: ["STOP", "Zero", 0 ],
    0x15: ["ISZERO", "VeryLow", 0 ],
    0x34: ["CALLVALUE", "Base", 0 ],
    0x39: ["CODECOPY", "VeryLow", 0 ],
    0x52: ["MSTORE", "VeryLow", 0 ],
    0x57: ["JUMPI", "High", 0 ],
    0x60: ["PUSH1", "VeryLow", 1 ],
    0x80: ["DUP1", "VeryLow", 0 ],
    0x5b: ["JUMPDEST", "Special1", 0 ],
    0xf3: ["RETURN", "Zero", 0 ],
    0xfd: ["REVERT", "Zero", 0 ],
}
gascosts = {
    "Zero": "tier0Gas",
    "Base": "tier1Gas",
    "VeryLow": "tier2Gas",
    "High": "tier5Gas",
    "Special": "tier7Gas",
    "Special1": "tier0Gas",
}
tiers = {
```

```

        "tier0Gas" : 0 ,
        "tier1Gas" : 2 ,
        "tier2Gas" : 3 ,
        "tier5Gas" : 10 ,
        "tier7Gas" : 0 ,
    }
    gas = 0
    newgas = 0
    cnt = 0
    while (i < len(b)):
        byte = b[i] + b[i+1]
        byte = int(byte , 16)
        if inst[byte][0] == "PUSH1":
            i = i + inst[byte][2] * 2
            byte1 = b[i] + b[i+1]
            print "(%d) %s gas %s" % (cnt , inst[byte][0] + ' 0x' + byte1 ,
                tiers[gascosts[inst[byte][1]]])
            gas = gas + tiers[gascosts[inst[byte][1]]]
        elif inst[byte][0] == "MSTORE":
            print "(%d) %s gas %s" % (cnt , inst[byte][0] + ' ' , tiers[gascosts[inst[byte][1]]] + 9)
            print 3 * 3 + 3 * 3 / 512 #GasCosts::memoryGas=3 GasCosts::quadCoeffDiv=512
            size=3
            gas = gas + tiers[gascosts[inst[byte][1]]] + 9
        elif inst[byte][0] == "STOP":
            break
        else:
            print "(%d) %s gas %s" % (cnt , inst[byte][0] + ' ' , tiers[gascosts[inst[byte][1]]])
            gas = gas + tiers[gascosts[inst[byte][1]]]
            print "Gas total so far is %d" % gas
            if inst[byte][0] == "JUMPI":
                newgas = gas
                if inst[byte][0] == "REVERT":
                    gas = newgas
                if inst[byte][0] == "JUMPDEST":
                    gas = gas + 1
                if inst[byte][0] == "CODECOPY":
                    gas = gas + 6
            print 3 * ((53 + 31) / 32) #memoryGas = 0 wordGas = 6 _
            multiplier/GasCosts::copyGas=3
            i = i + 2
            cnt = cnt + 1

    print "Total gas consumed is %d" % (gas)
    f = open("binr.data")
    all = f.read()
    b = json.loads(all)
    b = b['contracts']['b.sol:Test']['bin-runtime']

```

```
print (len(b)/2 * 200) + gas
```

Output

```
(0) PUSH1 0x60 gas 3
Gas total so far is 3
(1) PUSH1 0x40 gas 3
Gas total so far is 6
(2) MSTORE gas 12
9
Gas total so far is 18
(3) CALLVALUE gas 2
Gas total so far is 20
(4) ISZERO gas 3
Gas total so far is 23
(5) PUSH1 0x0e gas 3
Gas total so far is 26
(6) JUMPI gas 10
Gas total so far is 36
(7) PUSH1 0x00 gas 3
Gas total so far is 39
(8) DUP1 gas 3
Gas total so far is 42
(9) REVERT gas 0
Gas total so far is 42
(10) JUMPDEST gas 0
Gas total so far is 36
(11) PUSH1 0x35 gas 3
Gas total so far is 40
(12) DUP1 gas 3
Gas total so far is 43
(13) PUSH1 0x1b gas 3
Gas total so far is 46
(14) PUSH1 0x00 gas 3
Gas total so far is 49
(15) CODECOPY gas 3
Gas total so far is 52
6
(16) PUSH1 0x00 gas 3
Gas total so far is 61
(17) RETURN gas 0
Gas total so far is 61
Total gas consumed is 61
10661
>solc --gas b.sol
contract Test {}
^-----^
```



```

===== b.sol:Test =====
Gas estimation:
construction:
    61 + 10600 = 10661

```

The solidity compiler displays the gas our contract will use when the option `--gas` is passed. The gas calculated by our program and the solidity compiler is the same.

As always, let's look at changes made in the code when it comes to calculating gas. It's not easy to compute how much gas a contract will consume. Some opcodes will need more gas, some less, some not at all. All this depends upon the opcode.

Ethereum has created categories of gas consumption like Zero, Base, High etc. In our list, with every opcode, the gas consumption category is also stored. It is just an indication of whether it is a gas guzzler or not and not the actual consumption. Then, one more layer of abstraction is added. Each category in the list is now placed in a certain tier. This mapping is done by the `gascosts` dictionary. Finally, the tiers have an actual gas value associated with them. The opcode `JUMPI` has a gas category of high, its tier is `tier5gas` and it consumes 10 gas units.

Now comes the complications of determining the actual gas consumed. For most instructions, the cost of gas is computed by reading one key from another using the three dictionaries. We read backwards, the `inst` dictionary gives the gas category, the `gascosts` gives the tier and finally the tiers give the actual gas cost.

The opcode `MSTORE` is more complex, we also compute the `memorygas` which in this case is 9, total gas is 12. Then for the `CODECOPY` opcode, a complex calculation is used to calculate the gas which is 6. We are dealing with a `wordGas` and a `memoryGas`.

For a `REVERT` opcode, first the gas is saved and then with a `JUMPI` opcode, the gas is lowered to the stored value. It took us a long time to arrive at this code, so we understand your confusion.

The second gas is computed by multiplying 200, a constant with the length of the bytecodes taken while running the `bin-runtime` command. This gas is called a `codeDepositGas`.

The gas we calculate is not in ether units. There is a very good reason for it. Let's assume that we pay 10 ethers for executing one `PUSH1` opcode or instruction. There was a time when 1 ether cost less than a dollar. We could pay 10 dollars to execute a `PUSH` instruction. Today 300 dollars would buy you an ether. We would have to pay \$3000 to execute the same `PUSH` opcode. Nobody in his/her right mind would execute a push opcode for fun. We cannot change the gas costs each time the price of ether moves up or down.

The solution is very simple. Let's introduce a new currency called gas, for want of a better name. When we started, 1 gas bought say 20 ethers. The push instruction is now in gas units, it costs $10 * 20$ or 200 ethers to execute a push instruction. We prefer using such large values than using a wei.

When the price of ether jumps through the roof, all that we do is reset the gas to ether ratio. 1 gas now buys 0.01 ethers and not 20. The amount we now pay for a push instruction is $10 * 2$ or only 20 ethers. At 1 ether to 1 dollar, if 200 dollars were used for a push instruction, now it is $10 * 0.01$ or 0.1 ether or 30 dollars. The price of Ethers shoots up, but we pay less gas to execute my code. In Bitcoins, we pay a sort of fixed price to the miner. Here it all depends upon the amount of computation performed. The greed for gas opcodes can cost you more gas. If you do not have the right amount of gas, like us in a past example, the smart contract will not run.

Ethereum is more complex than Bitcoin. An in-depth knowledge is required to make Smart Contracts secure. The developers of Ethereum are not helping in any way. We hope to fill in the blanks in our next book on Ethereum contracts security. All good things must come to an end.

CHAPTER 39

What is a Bitcoin and a Blockchain

Shakespeare once said “what’s in a name? That which we call a rose by any other name would smell just as sweet”.

So, What exactly is a Bitcoin and What is a blockchain?

Mr. Gavin Andresen, who was the lead developer of Bitcoin and is considered to have the last word on Bitcoin published a blog in Feb 2017 and it had the definition of Bitcoin. Let’s look at the technical definition of Bitcoin in the words of Mr. Andresen and scrutinize it.

>Bitcoin is a ledger.

There are a trillion definitions of the word ledger, a book or a computer file, or an accounting definition.

>Bitcoin is a ledger of not-previously spent.

Obviously, we cannot spend money that we do not own. So, how do we determine the amount of money we own. In the Bitcoin world, the UTXO set does not store a running account of our Bitcoin balance but it keeps checks and balances of unspent outputs. The wallet file monitors every Bitcoin transaction and keeps a running balance of our Bitcoin addresses only. But, there is no concept of a global wallet.

Ethereum stores a balance of every Ethereum address in the state machine.

It does not matter how an entity keeps track of our right to spend the crypto currency, there must be a system in place to ensure that we do not spend what we do not own.

>Bitcoin is a ledger of not-previously spent, validly signed transactions.

There must be some technology in place to claim ownership of Bitcoins. This can be achieved by signing with a private key and using its public key to verify the credentials of the private key without possessing the private key. One can use Elliptic Curve Cryptography or RSA (to create signatures) or in the future, Schnorr signatures. In the far future, quantum bits are likely to be used for signing. The actual signature and the method or algorithm is not important here. What’s important is to prove ownership of Bitcoins. We don’t have to use cryptography to sign a transaction.

>Bitcoin is a ledger of not-previously spent validly signed transactions contained in a chain of blocks.

A transaction is grouped together in a block and the blocks are chained or linked to one another. There is no hard and fast rule that the blocks must exist physically on disk. They could be stored in a SQL database or in a leveldb database like Ethereum. Physically store them in any way, but logically they should be seen as transactions in a block, each block connected to the next block and the previous block. The hashes are not significant while chaining the blocks. This connection can be established using physical block numbers for all that we care.

>Bitcoin is a ledger of not-previously spent, validly signed transactions that begins with the genesis block (hash 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f).

The genesis or the first block must be hardcoded in the code. There must always be a known starting block. The Bitcoin ecosystem uses a certain hash value for representing the genesis block, it could be another hash for another currency, this hash must be fixed and known in advance.

>Bitcoin is a ledger of not-previously spent, validly signed transactions that begins with the genesis block (hash 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f)

follows the 21-million-coin creation schedule.

In Bitcoin, there is an upper limit on the total number of Bitcoins that can be mined. Other currencies may have no upper limit or use one smaller than the Bitcoin limit. The demand here is to have rules in place on the number of coins that can be created. Moreover, there is complete silence on the miner's reward and on the time interval, when creating these coins.

>Bitcoin is a ledger of not-previously spent, validly signed transactions that begins with the genesis block (hash 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f)

follows the 21-million-coin creation schedule, and has the most cumulative double-SHA256-proof-of-work

By this definition, there must be some proof of work that determines which miner calculated the double SHA-256 hash, and ultimately allowed the block to be added to the blockchain.

There is a possibility in the distant future that there will be Proof of Stake in Ethereum, Intel's Proof of Elapsed Time (sound's good POET), and zillions more. The idea is to make sure that people or complete strangers, world over come to a common understanding of the state. Everyone must agree that Vijay Mukhi is the proud owner of 10 Bitcoins.

The reason behind evaluating the above definition is to get a clear understanding on the technology that drives Bitcoin, it is called blockchain. If there is no consensus on a definition of bitcoin, it looks seemingly difficult to agree on what comprises a blockchain and whether Corda can call itself a non-blockchain. Some of the biggest financial players in the world created a company called R3, which released a product called Corda in October 2017. Yes, this is a final release not a beta or Proof of Concept.

The definition does not lay any emphasis on the relevance of the Merkle hash. Plus, there is ambiguity on the computation of the hash of all the transactions, even Ethereum computes the hash in its own way. The above definition insists we use a SHA hash but we disagree, as other future hashes will be more secure.

In the definition, there is no clarity on whether a transaction must be broadcasted to the whole wide world or only to interested parties. When my banking transaction travels from my bank to another bank, there is no reason for the world to see it. The debate is now on public vs private blockchains.

The definition is also silent on the number of people who can do the proof-of-work. Can we agree on 6 trusted parties to find the hash or must we all do it ourselves? As is, these days, the number of miners in the community is shrinking colossally. Someday, we will be left with say only 25 large miners. The independent miner was long dead. In theory, there cannot be a central authority that controls a crypto currency.

There are no rules to be followed while creating a Bitcoin address other than it being unique across time and space. The format of a transaction is not important as Ethereum has no formal concept of an input and an output. The index folder is important only because Bitcoin uses blk and rev files to store data. Remove these files and the index folder has no role to play, but something else comes in. For me, the chainstate folder is Bitcoin, the index folder is a helper.

The concept of a wallet is a must, not as a wallet in literal terms, but a physical file. We are silent on the use of Hardware wallets.

The peers.dat file looks redundant but then there must be a secure way of downloading transactions from a trusted source.

There is a desperate need for creating categories/slabs on fees payable to a miner to include the transactions. It may be a small issue but someone must write down the rules and standardize on the fees payable to a miner.

There will soon be many more script types and opcodes in Bitcoin. Ethereum smart contracts are at the other extreme in writing flexible code. The sky is the limit. We must have variations of the above principles so that we can define a blockchain.

There are too many questions left unanswered. It is a very complex problem at hand so people define a blockchain as they deem fit. Defining a blockchain is like defining Beauty, it lies in the eyes of the beholder, to quote Shakespeare once again.

As people, we are lost when it comes to defining a Bitcoin and a blockchain. Too many variables. One standard definition that pleases all will never be achievable. Hence, we end up with zillions of definitions.

Ethereum calls itself a state machine. Ethereum and Bitcoin are as different as chalk and cheese. But both are called blockchains. Finally, hyperledger, another consortium of very big people defines its existence as an umbrella of blockchains. They admit that there will never be one blockchain.

This reminds us of one million and six blind men and the elephant. Each saw something different but they were also spot on.

We still do not have a precise definition of a blockchain and we believe nobody has one either.

J P Morgan Chase is one of the largest bank in the world having assets worth 2.5 trillion dollars on its balance sheet. If that's not enough to impress you then handling 25 trillion dollars of Other People's Money will. That sort of money can buy you the entire Solar System. The present Chairman and CEO of J P Morgan Chase is Jamie Dimon who is known for the role he played in getting Citi and Travelers together, along with a noted financier Sanford Weill. When Jamie made a cardinal error of not promoting Weill's daughter, he got fired. Then he joined Bank One, the sixth largest bank in the US. This bank was taken over by J P Morgan Chase.

Jamie once said "Bitcoin developers would try and eat our lunch". He should have added dinner also.

If J P Morgan Chase can put their name and weight behind an Ethereum enabled product called Quorum, then the least we can do is write a few pages on it. What we have learned in our 40 years in the technology world is that pedigree counts. Big money backing something always gives a winner.

But, first the future of Bitcoins.

There is a raging debate on whether Bitcoin should be a payment layer or a settlement layer. A payment layer implies acting like another currency. And here is where the block size debate starts. If the block size is increased to say 100MB, then in the same ten minutes, 100 times more transactions can be confirmed. But there will be a wait of 10 minutes minimum for a confirmation. So, no buying coffee or groceries or shopping till then, we have seen that a delay of 30 seconds has flared up people greatly. Bitcoins can never replace a credit card ever.

Then comes the debate of the Bitcoin blockchain crossing 200GB in size in the year 2017. Once again block size does not matter. More the transactions using Bitcoins the greater the number transactions in the blocks, greater the size of the blocks. It makes no sense for every tom, dick or harry to store the blockchain on their local machines. Today, not everyone dreams of being a miner as you need to very deep pockets. So, why should everyone store the blockchain on their puny machines.

Finally comes the Bitcoin share price. In two months, it goes from 2500 dollars to 5000 dollars and then back again to 2500 dollars. In November 2017 it is holding steady at \$6000. Too much volatility. So, as an investment hedge fund, it's a bad idea.

In my view, Bitcoin as a settlement layer is a perfect fit. All transactions can take place on some sidechain and then Bitcoin can be used as a layer to guarantee these transactions. This will enable transactions to cost as low as one

millionth of a dollar. It costs an arm and a leg to send money to another country. The blockchain will make sure that the monopoly of bankers is broken, their pay checks reduced by one tenth and the extortion tax that we pay to the financial community will be wiped out once and for all. There are too many inefficiencies in the monetary ecosystem which make money very expensive. Blockchain can act as a sword in removing all hurdles that stand in the democratizing of money.

The Ethereum blockchain has absolutely nothing in common with the Bitcoin Blockchain.

Over to Ethereum.

We prefer Ethereum over Bitcoin as Ethereum is more programmable than Bitcoin. You cannot write a smart contract in Bitcoin today but maybe next year, who knows. Our view is that Bitcoin will always leave ether far behind in the crypto-currency sweepstakes. Also, products like Quorum, the privately created Ethereum of J P Morgan Chase will be the chosen one that businesses will use.

Over to Quorum.

Quorum sits on top of geth, the Go written standard Ethereum client. Quorum adds an extra layer above geth, it does not change geth in any way. Anyone can join as a Ethereum node but with Quorum, you must have prior permission.

In our view, this is the most sensible thing to do as I would not want to disclose my transactions with different banks. Plus, you do not need every Citizen Joe on the planet joining in and leaving the blockchain at their whims and fancies. For businesses, a permission based blockchain is most suitable, not an open one, which is currently being used by nearly every crypto currency.

Though there is an advantage of having a free hand, as the blockchain can then be verified by anyone, there is no separate central authority. Quorum works at two levels for gaining consensus, one is with public data and the other one for private data. This indicates that Quorum has two types of transactions, the normal Ethereum-like public transactions and the other is private transactions for private users.

Financial data is very heavily regulated and therefore the regulator wants to see all the data in real time. This applies only to the regulator and no one else. Everyone can verify the integrity of the blockchain but everybody must not see the transaction details.

The biggest problem with consensus is that it takes time. Nobody wants a 10 minutes Bitcoin consensus. Even Ethereum blocks takes 15 to 20 seconds to confirm. In the case of Quorum, you get Visa type time confirmations and volumes of transactions as well.

The Quorum blockchain is a Ethereum blockchain with data privacy. You can download the Quorum blockchain and create a private blockchain. No permission required here. So more smart contracts created, the better for J P Morgan. One article compared Quorum to Apple IOS, where you are encouraged to build apps.

One fallacy is that all Bitcoin transactions are in public domain since the blockchain in Bitcoin is open to all. This is true. But, there is always a but, no one knows who creates these Bitcoin transactions. If each time a different Bitcoin address is used, which is advisable, then even God can't trace it back to the source.

Quorum has also partnered with Zcash, which offers more anonymity than Bitcoin. The Secure Socket Layer (SSL) encrypts the data on the internet. Now SSL will be replaced by ZSL, the Zero Knowledge Secure Layer. Even hardened cryptographers cannot make sense of Zero Knowledge proofs.

The basic premise is that a blockchain between say 10 large business houses and their suppliers and customers will be by far bigger than most public blockchains. That's why the focus in Quorum is on incorporating robust security features to avoid any breach to their data privacy wall. After all, no company in the world would like to share confidential data and pricing patterns to anyone. Everyone's interests would surely be on their competitors price list. Businesses need a public blockchain where their data is private.

Zcash makes everything invisible on their blockchain, even the amount of the transaction is not made public. J P Morgan is now making legitimate business transactions invisible. But the regulator can see it all. The best of both worlds.

Quorum is also part of the Ethereum Enterprise Alliance EEA which has a very important agenda. There are too many smart contracts out there, no two smart contracts talk to each other. The EEA will make sure that all Ethereum smart contracts can communicate with each other otherwise there will be plenty walled smart contract that cannot communicate with each other.

If you ever visit the offices of J P Morgan Chase, they have an excellent cakeshop which also has a GUI for creating smart contracts. No command line interfaces. We tried out their GUI program written in Java and it works.

In our view, businesses will build on Ethereum and create private networks. There are many other kids on the block. Blockchain startup, Chain roped in Visa to create a payment network. Corda by R3Cev does not call itself a blockchain. Then there is the big elephant in the room, Hyperledger by the Linux Foundation which calls itself the umbrella of blockchains.

As we have said earlier, a blockchain is successful only when the entire business ecosystem gets together to use it. If not, then stick to conventional apps and not DApps.

With everyone pushing the blockchain train, it has to go someplace. But giants like Google and Facebook are busy with AI, they have nothing to do with Blockchains. The blockchain is being driven by the financial institutions and head honchos of different countries.

The darkest hour is the hour before dawn. There was silence in the bitcoin world that lasted most of 2016 and the first seven months of 2017. From August 2017, the blockchain sun is seen rising. Happy sunny days for all of us who stayed back and did not lose hope.

Towards the end, I would like to say that I am betting on Smart Contracts on both Ethereum and yes, Bitcoin if it ever happens.

CHAPTER 40

AI and Blockchain – Never The Twain Shall Meet

Artificial Intelligence or AI as widely known comes with many names, such as Deep Learning, Neural Networks, etc. We are caught in a dilemma here as there are two sets of people on the internet talking about AI and Block Chain together saving the world. Though we can't make up our mind on which one will change the world, but we do agree that presently, there are only these two options in the technology space.

This small chapter on Artificial Intelligence or AI is to determine if we can use AI and Block Chain together, in the same breath. We have recently concluded a non-technical book on AI and hence this sampler. Still, we are unable to figure out which of the two, AI or Blockchain can be crowned as the king of the difficulty hill.

Back to our roots. The only way to understand AI is by running code. Clarifai is a new entrant that competes with the giants of AI like Google, Facebook, Amazon, Microsoft, IBM and everyone else including a Ford and an Uber. Before we run our first program, we must install some libraries and follow certain rules.

```
sudo pip install clarifai
```

The first step is to install the clarifai python module which does nothing magical, it simply sends a web request to the clarifai webserver. Clarifai must put money on the table to hire expensive AI people so it demands having an account on www.clarifai.com with the credit card number (optional), even though there exists a generous free tier.

On the developer page, Clarifai gives us a big number or an API key to uniquely identify ourselves. Place this key in an environment variable as

```
export CLARIFAI_API_KEY=1234abcd
```

We are assuming our API key is 1234abcd, which is not true.

When we create an empty Clarifai object as seen in the program below, this API KEY is sent over the Internet to identify ourselves.

```
ch4001.py
from clarifai.rest import ClarifaiApp
from itertools import islice
import os
image = './vijaymukhi.jpeg'
app = ClarifaiApp()
model = app.models.get("demographics")
p = model.predict_by_filename(image)
dict = p['outputs']
li = dict[0]
print "\n%-25s %s" % ("Age" , "Probability")
for i in islice(li['data'][0]['regions'][0]['data']['face']['age_appearance']['concepts'],3):
```

```
print "%-25s %0.2f" % (i['name'], i['value'])
print "\n%-25s %s" % ("Gender", "Probability")
for i in li['data']['regions'][0]['data']['face']['gender_appearance']['concepts']:
    print "%-25s %0.2f" % (i['name'], i['value'])
```

Output

Age	Probability
61	0.59
60	0.59
63	0.57

Gender	Probability
masculine	0.96
feminine	0.04

In this program, we want the AI model to disclose our age and gender. Clarifai named this model demographics. It has many more model names that return different entities. The get function with the model name creates an instance of the required model. The model variable is given this handle.

The same model object is then used to pass the image. This function, predict_by_filename will read, in this case file vijaymukhi.jpeg from the local folder and send it over to the server. Please choose your own picture and give its filename. Size and shape of the image does not matter one wee bit.

This same function waits for an answer from the Clarifai server and then returns a json formatted object as a Python dictionary. We extract the data in the outputs key and look at the first member in the list. It is a nested json object that has the age and the probability that the age is correct. Then taking another path, we get our gender. The most difficult part of AI code is figuring out the nested json structures.

So, just by giving a picture of our face, the AI model can reveal our age and gender. We are 2 months short of 60, and AI is spot on. This looks simple as we are using an AI model created by someone else.

```
ch4002.py
from clarifai.rest import ClarifaiApp
from itertools import islice
import os
image = './food.jpeg'
app = ClarifaiApp()
model = app.models.get("food-items-v1.0")
p = model.predict_by_filename(image)
dict = p['outputs']
li = dict[0]['data']['concepts']
print "\n%-25s %s" % ("Object", "Probability")
for i in islice(li,10):
    print "%-25s %0.2f" % (i['name'], i['value'])
```

Output

Object	Probability
pizza	1.00
tomato	1.00
cheese	1.00

crust	1.00
sauce	0.99
dough	0.99
mozzarella	0.99
pepperoni	0.99
salami	0.98
meat	0.96

Now we take a very different image, food.jpeg which is a picture of a pizza. Use Google to find any picture with food in it. The model name is now, food-items-v1.0. This AI model takes in an image of food and gives out a list of food items present in it. It does not stop there, it goes further to give a probability value of every ingredient as well. In our case, it found pizza, tomatoes, meat etc.

How does an AI model work its magic? Read on.

What follows is the heart and soul of an AI model. The AI model is given an image and then the image parts are individually described and defined to it. For example, a pizza is described with its ingredients, as in dough, cheese and tomato etc. These descriptive words are called labels or categories. This process is called training an AI model. In real life, there are more than 10 million images. Once trained for the images, the models can then identify the image, be it food items or the age of person or whatever. Clarifai has different AI models for different categories.

This is very different from what Blockchain's do.

We now look at some different types of demos which showcase what an AI model can achieve.

<https://how-old.net/>

This web page is owned by Microsoft. Chose an existing picture or upload your own picture. The AI engine will tell you the age of every face in the image. It is similar to program ch4001.py

<https://www.kairos.com/demos>

Kairos is a new AI startup. Skip to the third or the last video demo where the video shows all the human emotions on a face in real time. It can be used as a lie detector, so job aspirants beware.

<https://indico.io/demos/clothing-matching>

A new startup Indico for fashion and more, especially for people with a bad dressing sense. Click on a shirt on the first list of images and see what pants match at the bottom set of images.

<https://www.clarifai.com/demo>

Clarifai has many demos on this webpage. Chose an image and a model and see how accurate Clarifai models can be. The model NSFW or Not Safe For Work figures out if an image is safe for work or not. In English, it means does the image have pornographic content.

<https://conversation-demo.mybluemix.net>

This is from IBM. You talk to a car in the English language and it will function as per your instructions. Generally, there is no human being at the other end of a chat.

<https://www.google.com/intl/en/chrome/demos/speech.html>

This one is from Google and the Chrome browser. Its depicts that an AI generated voice is undistinguishable from a

human voice. When you talk to someone on a phone, it is difficult to guess if it is a human or an AI bot. If it is smart, then you are undoubtedly interacting with an AI model.

<https://console.aws.amazon.com/rekognition/home?region=us-east-1#/face-comparison>

This is Amazon. You use two pictures. It will locate the face in the first image within the second image, which has pictures of multiple faces. You cannot hide in a crowd anymore.

Now that we have used AI models built by others, let's build some AI models. Building AI models is extremely difficult, it is not for the fainthearted. This chapter is trivial compared to an AI book, which we will never write.

The biggest player in this space is Google and they have a Python library called Tensorflow. Facebook another biggie, has PyTorch and Caffe2, Amazon has mxnet and the list goes on.

We use Python libraries, bearing in mind that building AI models on their own is very difficult. There is a Python library called Keras that sits on top of Tensorflow and Theano, supposedly the first AI library ever. We write our code in Keras and then Keras converts this code to run on Tensorflow or Theano or whatever. The developer of this library now works for Google and Keras is now an official Google product.

Let's build the smallest Keras model so that we understand what goes into building AI models.

```
ch4003.py
from keras.models import Sequential
from keras.layers.core import Dense
from keras.optimizers import SGD
import numpy as np
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])
model = Sequential()
model.add(Dense(8, input_dim=2, activation='tanh'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer=SGD(lr=0.1))
model.fit(X, y, epochs=1000, verbose=1)
print model.predict(X)
```

```
Output
Epoch 999/1000
4/4 [=====] - 0s - loss: 0.0243
Epoch 1000/1000
4/4 [=====] - 0s - loss: 0.0243
[[ 0.00927633]
 [ 0.97312218]
 [ 0.97472531]
 [ 0.03415722]]
```

The germ of this program came from finding Keras code in Google which could perform an XOR. This program has nothing to do with building useful AI models but helps us get started. All data to Python based AI libraries is in the form of a numpy array which is a multi-dimensional list. They are commonly called matrices, now numpy array's and in AI, a tensor.

A matrix deals with all kinds of dimensions or shapes. Here we have only 4 data points which is all that a binary XOR

can handle. The first lesson of AI data handling is that whether the input data is an image or stock values, it is always passed using a numpy array. The X numpy type variable contains the input data, the data we feed our model. The code on the net does not use a small x for variable names, they call it X, the emphasis is on the capital X.

Then for every set of data we feed, we also hand over an answer to the AI model. AI models use the data and the answer and then decipher the future answers on unseen data. This variable is called small y and not capital Y. Confusing, so when to capitalize a variable?

All Keras models are of type Sequential. So, we build a Sequential model object called model which is purely a Keras concept, nothing to do with AI. A model is made up of different types of layers. This is like a house being made up of different types of floors.

The first layer we use is called a Dense layer. There are a large number of layer types which are used for different purposes. The parameter input_dim is assigned the type of data we feed our model. We have only two dimensions in our data. The activation parameter decides how the neuron will activate or fire. Once again too many options.

It does not stop here. Another Dense layer is added. The first parameter is the type of prediction or answer our AI model will return. This is the last layer, in our case. This implies that our AI model will answer in binary, 0 or 1, yes or no. The activation type is sigmoid which is used for boolean outputs. The first layer's activation type was tanh.

Finally, the model is compiled, another Keras addition.

Now the AI segment starts. The model must be trained and for this task there is a fit function. The verbose parameter displays the workings and hence its value is 1. The fit function is given the input data as well the corresponding answers, as the first two parameters. The fit function will feed the first set of data, values are 0 and 0. It will then try and adjust the number or weights associated with each neuron in the new AI model.

The AI will then compute the answer. It compares its answer with our answer. Next, it looks out for errors or losses. The loss function helps in computing this loss and the optimizer function determines a way to obtain the correct answer with a minimal of fuss. This loop goes on 1000 * 4 times, where 1000 is our epoch. An epoch refers to how many times. We see the calculated loss at each epoch.

At the end of the fit function, we have an AI model that understands how to predict an XOR computation, in our specific case. The predict function predicts the answer. This AI model worked fairly well.

There is a touch of irony, we feed the AI model with a set of data and the answers. But, we cannot use the same data in AI and ask for predictions. The training and test data must be different.

```
ch4004.py
import numpy as np
from keras.models import Sequential
from sklearn.model_selection import train_test_split
from keras.layers import Embedding, LSTM, Dense
data = np.loadtxt("pima-indians-diabetes.data", delimiter=",")
X = data[:, 0:8]
y = data[:, 8]
Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, test_size=0.10, random_state=42)
print len(X) , len(Xtrain) , len(Xtest) , len(Xtrain) + len(Xtest)
model = Sequential()
model.add(Embedding(20000, 128))
model.add(LSTM(12))
model.add(Dense(1, activation='sigmoid'))
```

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.fit(Xtrain, ytrain, epochs=15, verbose=1)
s = model.evaluate(Xtest, ytest)
print("\ns:%f" % (model.metrics_names[1], s[1] * 100) )
print("%s:%f" % (model.metrics_names[0], s[0]) )
```

Output

```
767 690 77 767
Epoch 1/15
690/690 [=====] - 2s - loss: 0.6730 - acc: 0.6406
Epoch 15/15
690/690 [=====] - 1s - loss: 0.2551 - acc: 0.8971
32/77 [=====>.....] - ETA: 0s
acc:70.129871
loss:0.745672
```

Here we have some data on Pima Indians having diabetes or not.

```
1,85,66,29,0,26.6,0.351,31,0
8,183,64,0,0,23.3,0.672,32,1
```

We will not explain these columns because it is left for the AI model to learn from the given data. Our job is to create the model and then leave it to the AI model to learn by finding similar patterns. Just a hint, the last column is the answer, whether the indian had diabetes 1 or not 0.

A dataset is created using numpy to load the diabetes data from a file on disk. The first 8 columns have data which is fed in a numpy array variable called X. The last column is the answer and it is stored in a numpy variable, y.

There is a python library, sklearn that has a function called train_test_split. This function takes the original data and splits it into a training set and a testing set. People build AI models in python because of the large library support that other programming languages lack. The size of the training and test set is given in the parameter, test_size. The lengths of each set are displayed.

A Sequential model is created like before with three different layer types, an Embedding layer and a LSTM and finally a Dense layer. LSTM stands for Long Short Term Memory. This layer is a flag bearer for a Recurrent Neural Network or a RNN. For images, a Convolutional Neural Network or a CNN is used. Deep Learning is made of CNN's and RNN's. The above statement is true 99.99% of the time.

The fit function trains the model and thereafter the evaluate function is used to measure the success of our model. The point to be emphasized here is that one data set is used for training and a different set of data is used for testing. The data sets are different as chalk is from cheese. We get an accuracy of 70 percent which is pretty good for starters.

ch4005.py

```
from keras.applications import VGG16, imagenet_utils
from keras.preprocessing.image import img_to_array, load_img
import numpy as np
model = VGG16(weights="imagenet")
img = load_img('dog.jpg', target_size=(224, 224))
img = img_to_array(img)
img = np.expand_dims(img, axis=0)
```

```

img = imagenet_utils.preprocess_input(img)
p = model.predict(img)
P = imagenet_utils.decode_predictions(p)
for ( (id, label, prob)) in (P[0][0:2]):
    print "%s:%s:%0.2f%%" % (id, label , prob * 100)

```

Output

```

n02099601:golden_retriever:82.86%
n02099712:Labrador_retriever:15.42%

```

So, we have learnt that there are many layers and they have their types. So how do we figure out how many layers and what types are used in an AI model? How will we know which loss function or optimizer or activation function is used?

The simple answer is we will never know. AI is a 50-year old science but we have yet to come across any text that explains the internal workings of an AI model. Most people treat the inner workings of AI models as a black box. An AI model is simply a set of numbers called weights.

But, there is help at hand. There is a database and a competition called ImageNet. They have over 14 million images that have been categorized, labeled or given a meaning. There is a yearly competition that recognizes the AI model which is best at understanding the contents of an image. One of the previous winners has been a AI model called VGG. The source code of VGG model is available freely.

Keras allows us to instantiate a VGG model. Since we cannot run 14 million images to train this AI model, we create a VGG object. This will download the weights or numbers that comprise this model. In other words, it's like training the VGG model on our own computers. This VGG model is of type CNN.

Then we take a picture of a dog, you can take an image you like. This image is loaded from disk and resized. There is a standard pre-processing of the image to match the format expected by a VGG model. The predict function predicts the contents of our image with a probability. In our case, the AI model says it is a golden_retriever.

```

ch4006.py
from keras.applications import ResNet50 , InceptionV3 , Xception , VGG16 , VGG19
from keras.applications import imagenet_utils
from keras.applications.inception_v3 import preprocess_input
from keras.preprocessing.image import img_to_array , load_img
import numpy as np
MODELS = {
    "vgg16": VGG16,
    "vgg19": VGG19,
    "inception": InceptionV3,
    "xception": Xception,
    "resnet": ResNet50
}
for i in MODELS:
    print "Model Name is %s" % i
    if i == 'vgg16' or i == 'vgg19' or i == 'resnet':
        inputShape = (224, 224)
        preprocess = imagenet_utils.preprocess_input
    else:

```

```
inputShape = (299, 299)
preprocess = preprocess_input
model = MODELS[i](weights="imagenet")
img = load_img("dog.jpg", target_size=inputShape)
img = img_to_array(img)
img = np.expand_dims(img, axis=0)
img = preprocess(img)
preds = model.predict(img)
P = imagenet_utils.decode_predictions(preds)
for ( (ID, label, prob)) in (P[0][0:2]):
    print "\t%s:%0.2f%%" % (label , prob * 100)
```

Output

```
Model Name is vgg16
    golden_retriever:82.86%
    Labrador_retriever:15.42%
Model Name is inception
    Labrador_retriever:80.69%
    golden_retriever:4.06%
Model Name is xception
    Labrador_retriever:70.36%
    golden_retriever:8.50%
Model Name is resnet
    Labrador_retriever:80.71%
    golden_retriever:13.66%
Model Name is vgg19
    golden_retriever:45.02%
    Labrador_retriever:43.21%
```

Why stop at just one AI model, Keras comes with 5 of the world's most well-known AI models in the universe. Every AI model respond to the same image differently. AI models are generally not built from scratch. We choose the AI model that fits our specific case.

We have seen AI models that recognize objects in images. The same AI model can be trained to recognize a tumor in the lung. We use pre-trained AI models to recognize things in images. This science is called Transfer Learning, like Einstein once said learn by standing on the shoulders of tall men, not by standing on other people's toes.

We see no convergence between AI and Blockchain. The technologies of AI and Block Chain are like yin and yang or two different sides of the same coin. The only thing common between them is a fat paycheck and a terrible headache as they are equally demanding of intelligence. You cannot go wrong in learning one or the other.

If we had a choice between Blockchain and AI, we would choose Blockchain hands down. The only reason being that we like digging deep into any subject we evangelize. With Blockchain the source is the last word. In AI, there are layers within layers and finally we end up in a black hole. When the experts in AI themselves cannot figure out why the AI models work their magic, what can we mere mortals do.

Despite not knowing how AI models work at a fundamental level, we trust them to run our lives. It comes as no surprise that AI model builders make more money than the Blockchain ecosystem does.